Message Passing Interface (MPI)

Prof. Michele Amoretti

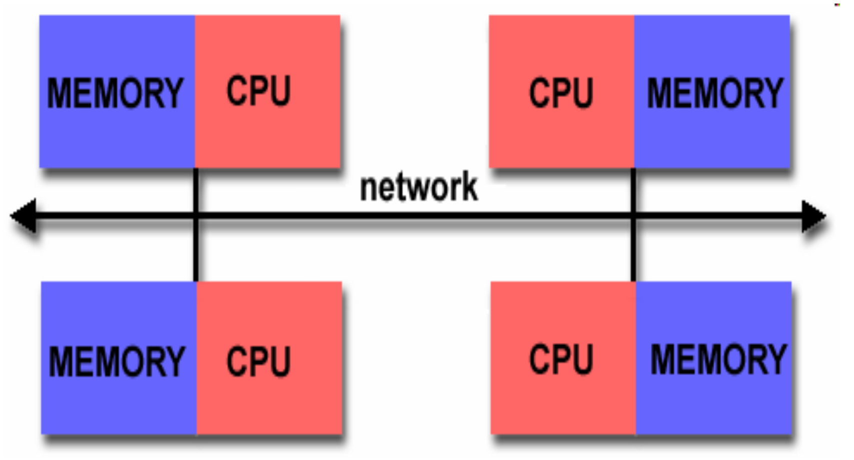
High Performance Computing 2022/2023

Summary

- Introduction
- Programming with MPI
- Point-to-Point Communication
- Derived Datatypes
- Collective Communication

Introduction

Distributed Memory



Distributed memory requires a communication network for the information exchange.

Each processor has its own local memory. Each memory has a separate, independent address space.

Read/write operations are local, for which there are no problems of cache coherence. To allow a task to access remote data, the programmer must explicitly manage the communication among tasks.

The corresponding programming model is called **message passing**. It dictates that each task can directly access only its local memory and must communicate with remote tasks to access remote data.

What is MPI?

A message-passing library specification
- message-passing model
- not a compiler specification
- not a specific product

For parallel computers, clusters, and heterogeneous networks

Full-featured

Designed to permit the development of parallel software libraries

Designed to provide access to advanced parallel hardware for
- end users
- library writers
- tool developers

What is MPI?

MPI is a specification for a library interface, not an implementation; there are multiple implementations of MPI.

MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, Fortran-77, and Fortran-95, are part of the MPI standard.

The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers.

MPI goals

Design an application programming interface (not necessarily for compilers or a system implementation library).

Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processor, where available.

Allow for implementations that can be used in a heterogeneous environment.

Allow convenient C, C++, Fortran-77, and Fortran-95 bindings for the interface.

Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.

Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.

Semantics of the interface should be language independent.

The interface should be designed to allow for thread safety.

MPI history

Version 1.0: May, 1994
Version 1.1: June, 1995
Version 1.2: July 18, 1997
Version 2.0: July 18, 1997
Version 1.3: May 30, 2008
Version 2.1: June 23, 2008
Version 2.2: September 4, 2009
Version 3.0: September 21, 2012
Version 3.1: June 4, 2015

Version 4.0: June 9, 2021

Version 5.0: work in progress

MPI Forum
http://www.mpi-forum.org

MPI 4.0

The standard includes:

- Point-to-point communication,
- Partitioned communication,
- Datatypes,
- Collective operations,
- Process groups,
- Communication contexts,
- Process topologies,
- Environmental management and inquiry,
- The Info object,
- Process initialization, creation, and management,
- One-sided communication,
- External interfaces,
- Parallel file I/O,
- Tool support,
- Language bindings for Fortran and C.

What is not included in the MPI standard?

- Operations that require more operating system support than is currently standard (for example, interrupt-driven receives, remote execution, or active messages)

- Program constructions tools

- Debugging facilities

MPI implementations

Several MPI implementations are available (e.g., MPICH and Open MPI).

- **MPICH** (C/C++, fully supports MPI 4.0)
https://www.mpich.org/

- **Open MPI** (C/C++, fully supports MPI 3.1, some functionalities of MPI 4.0)
https://www.open-mpi.org/

- **Intel MPI Library** (C/C++)
https://software.intel.com/en-us/intel-mpi-library

Open MPI

Open MPI is more efficient and portable than MPICH.

Compiling:
mpicc –o myprog myprog.c

Running:
mpirun [-np <num>] [-hostfile <hostfile>] <execfile>
where <num> is the number of processes, <hostfile> is the name of a file that lists the IP addresses of the machines, <execfile> is the name of the program to execute

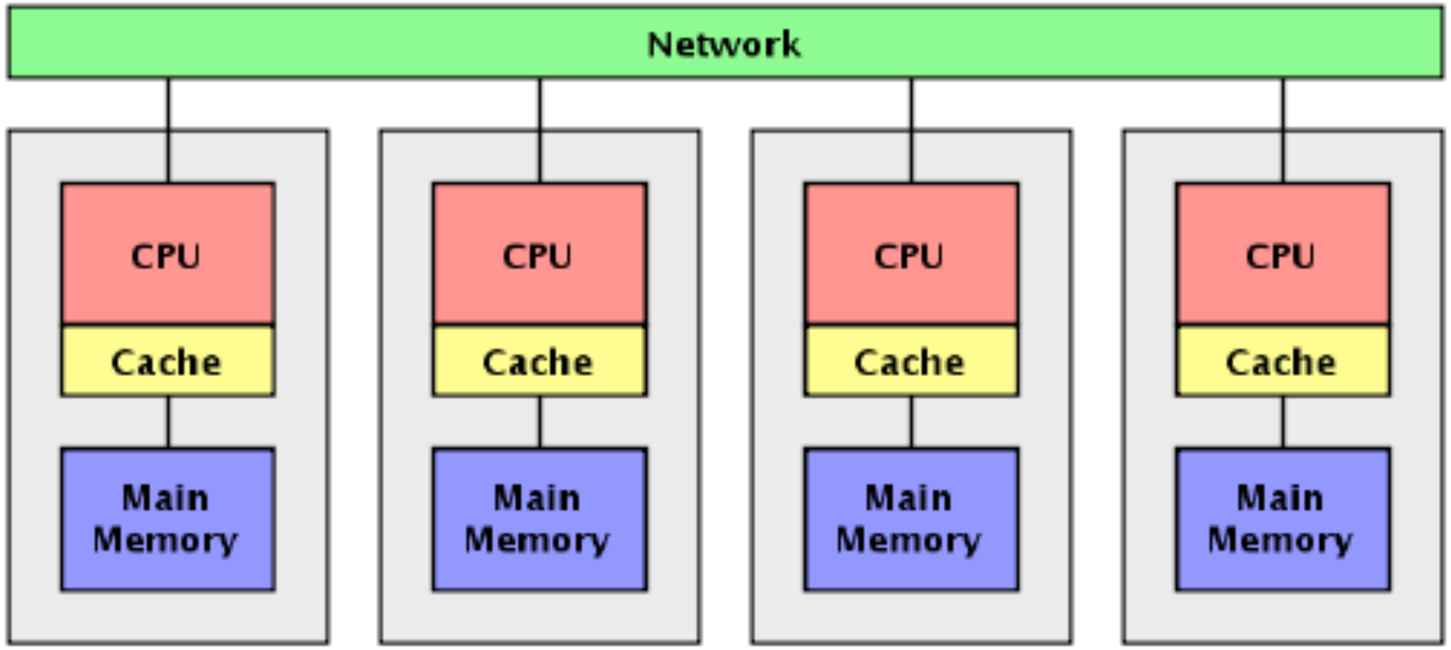
Hostfiles

Hostfiles are simple text files with hosts specified, one per line. Each host can also specify a default maximum number of slots to be used on that host (i.e., the number of available processors on that host).
Comments are also supported, and blank lines are ignored.

For example:

```
# This is an example hostfile.  Comments begin with #
#
# The following node is a single processor machine:
foo.example.com

# The following node is a dual-processor machine:
bar.example.com slots=2

# The following node is a quad-processor machine, and we absolutely
# want to disallow over-subscribing it:
yow.example.com slots=4 max-slots=4
```

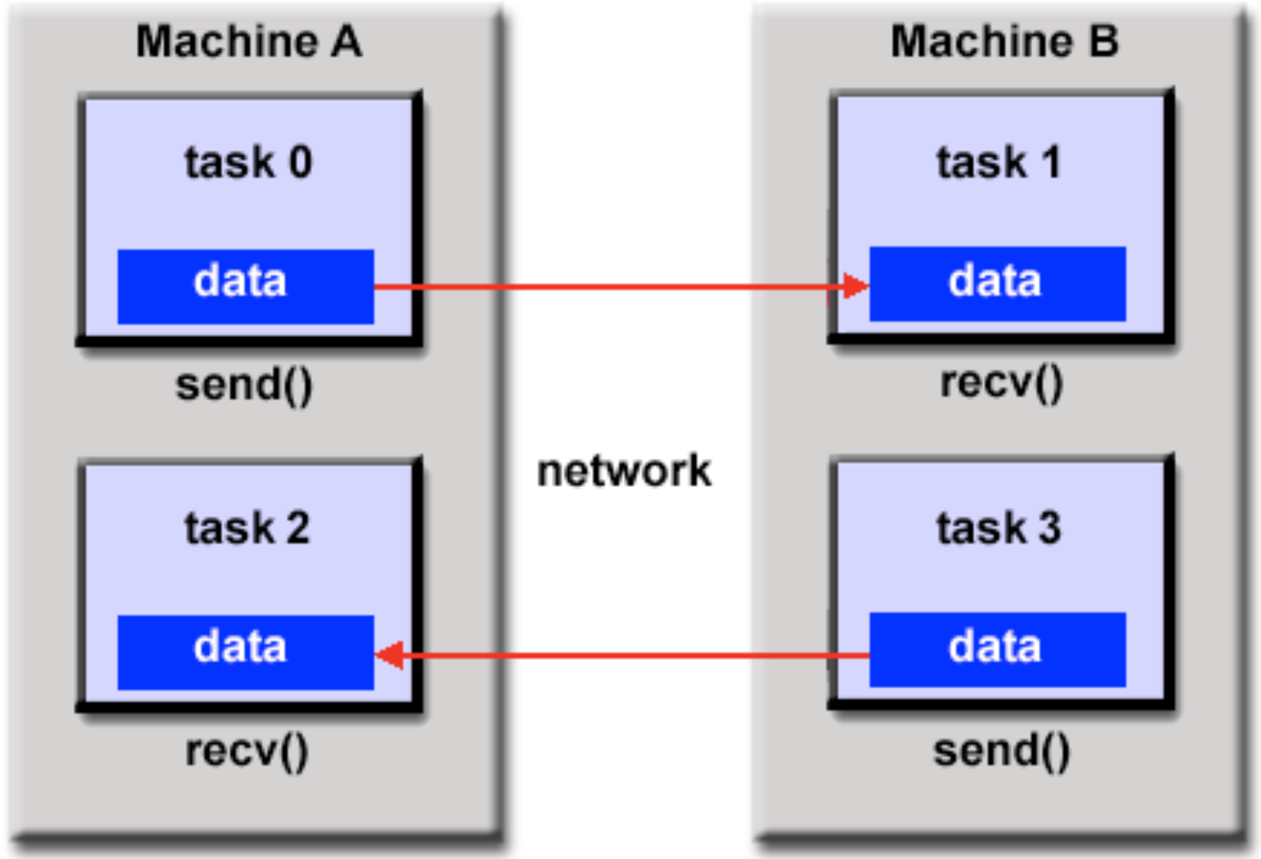
Message passing with MPI



Task = Process = Instance of running program

Message passing with MPI

Cooperation among processes is based on **explicit communication**.
A **sender** process and a **receiver** process exchange messages.



Message passing with MPI

Each process is an instance of running sub-program. Usually, the same sub-program is executed over different data sets (each execution being a process).

SPMD (single program, multiple data)

Each process is identified by an integer number, called *rank*, ranging from 0 to n-1, where n (size) is the total number of processes.

MPMD can be emulated with SPMD (using if-then-else constructs).

Messages

Messages are composed by two parts:

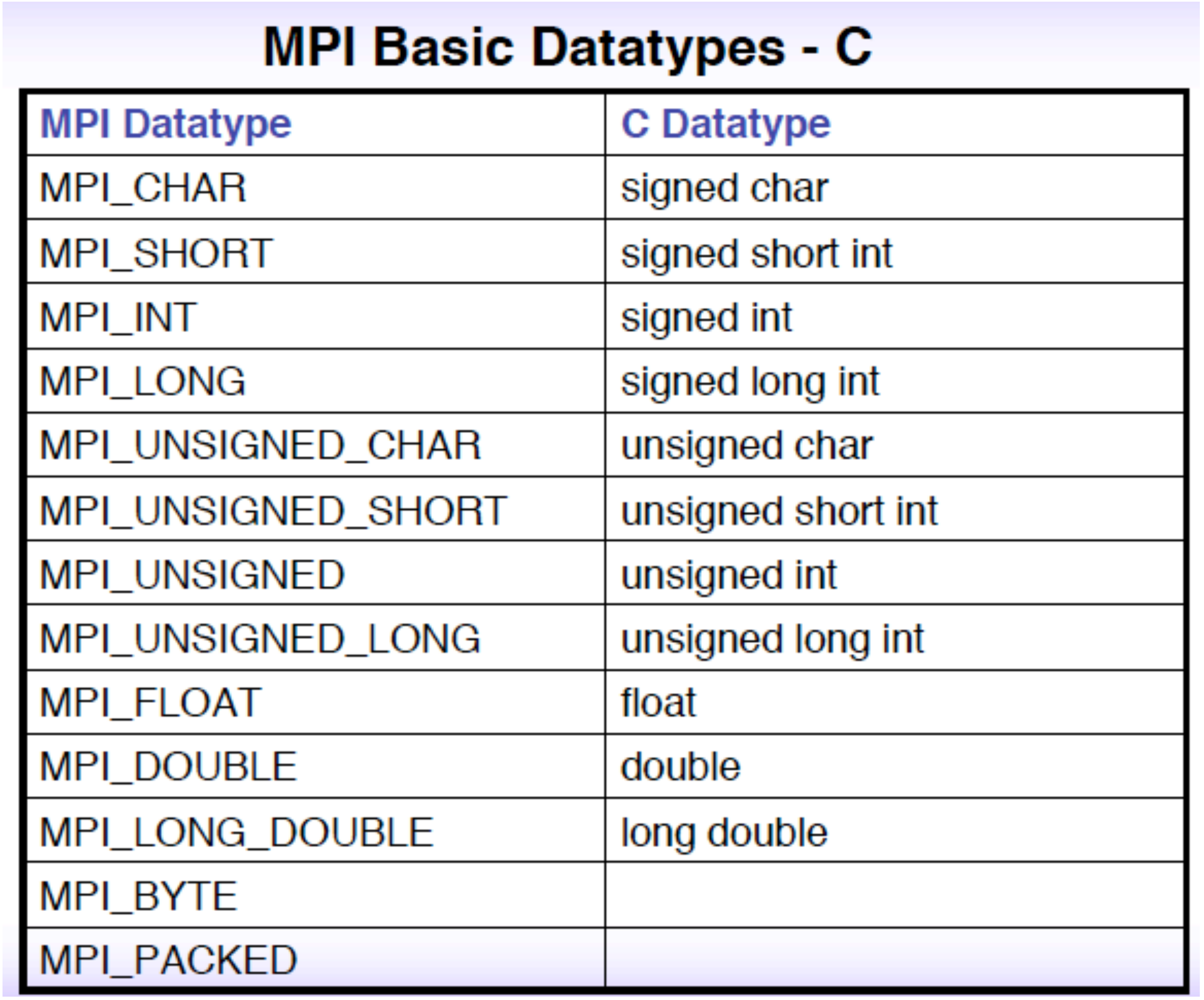
Envelope
source: rank of the sender
destination: rank of the receiver
tag: ID of the message (from 0 to MPI_TAG_UB)
communicator: context of the communication

Body
type: MPI datatype
length: number of elements
buffer: array of elements

Messages

- basic datatypes (corresponding to standard types of C or Fortran)

- derived datatypes (built from basic or other derived datatypes)



MPI Basic Datatypes - C

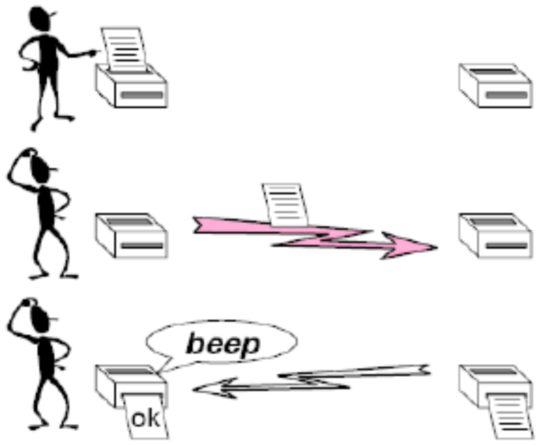
MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Point-to-point communication

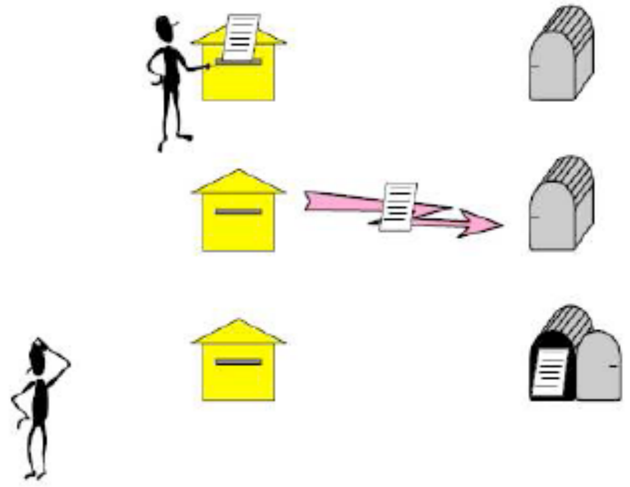
It is the most simple type of communication, involving only 2 processes (sender and receiver).

It may be:

- synchronous

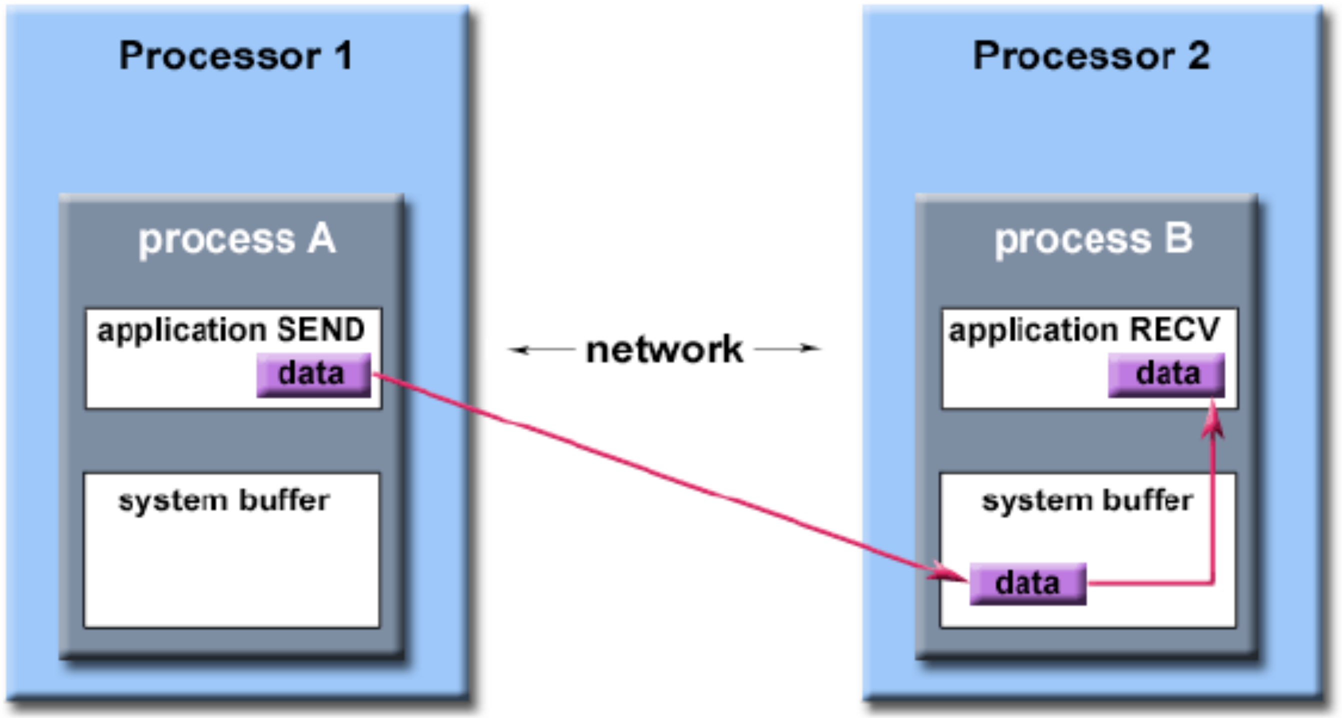


- asynchronous



Buffering

Buffering is implemented differently by each MPI library (the standard does not provides any related specification).



Buffering

Addressing spaces managed by programmers, for allocating variables, are called **application buffers**.

The **system buffer**:
- may exist at both sides: sender and receiver
- is usually limited
- has usually unpredictable behavior (not well documented)
- cannot be controlled by the application programmer

MPI also provides for a user managed **transfer buffer** (buffered send).

Operation modes

Some operations may cause the blocking of the caller.

Nonblocking operations allow the process to continue, immediatey after the call.
The process may **test** or **wait** for remote process completion (test), right after the nonblocking call.

Nonblocking op. + wait = blocking op.

Operation modes

Blocking:
- synchronous send
- asynchronous
  - buffered send
  - standard send
- ready send
- recv
- sendrecv

Non Blocking: same primitives, but the sender never blocks - it is necessary to check when buffers are reusable, when the communication has completed, etc. using
- test
- wait

Collective communication

This type of communication involves more than 2 processes (usually all).

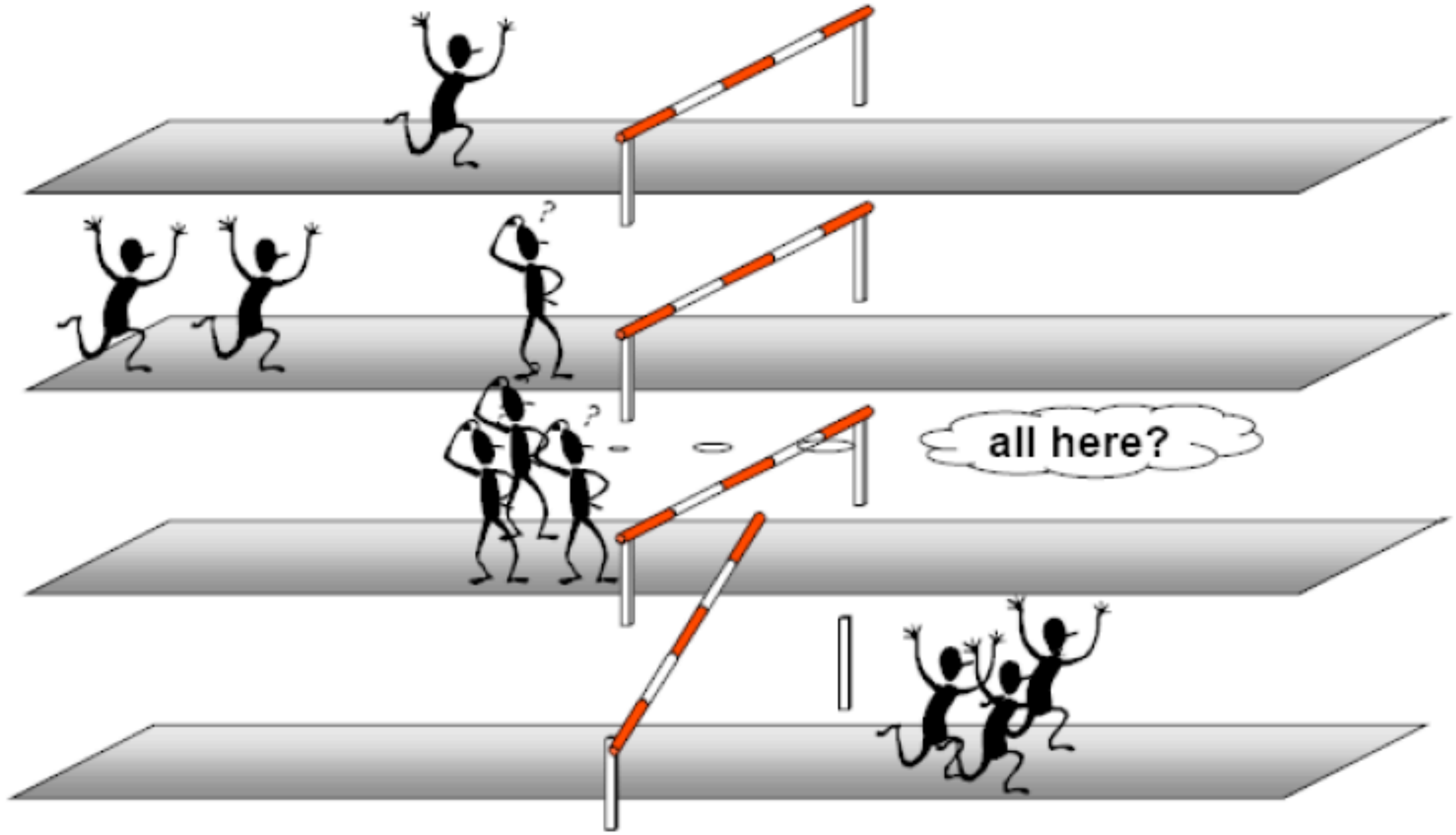
Barrier (for synchronization)

Data Movement (collective communications)
- Broadcast
- Scatter
- Gather

Reduction (collective computations)
- Minimum, Maximum
- Sum
- Logical OR, AND, etc.
- User-defined

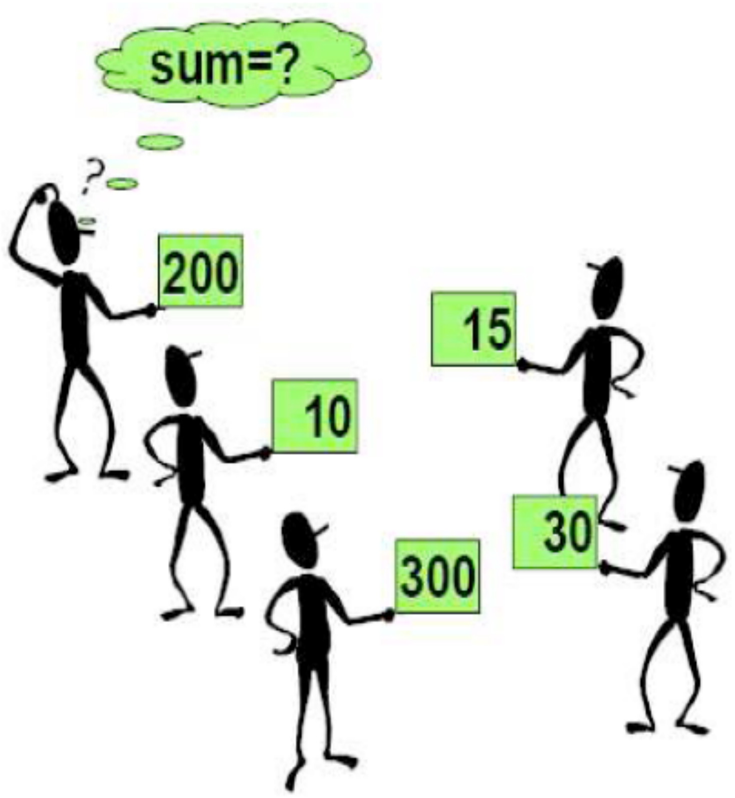
Collective communication

Example: using a barrier to synchronize processes



Collective communication

Example: using reduction to combine data from several processes and obtain a single result.



Programming with MPI

Functions in MPI

```
error = MPI_Xxxxx(parameter, ...);
MPI_Xxxxx(parameter, ...);
```

MPI_ is a reserved namespace for MPI constants and routines.

After the prefix, only the first letter is capitalized.

All MPI functions return an integer code.

Constant names are completely capitalized.

MPI header file

```
#include <mpi.h>
```

mpi.h is the standard header file for C.
mpif.h is the standard header file for Fortran.

The header file contains definitions, macros and function prototypes that are necessary for compiling MPI programs.

An **MPI handle** is a language-agnostic name for default communicators, standard datatypes, etc.

Communicators

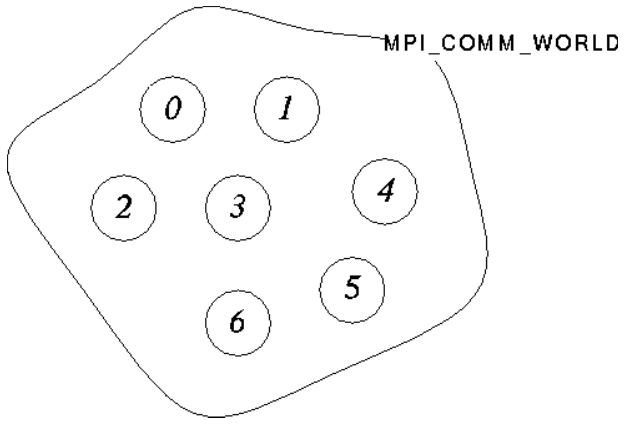
A communicator is a set of processes that can communicate with each other.

- has a name
- has a size (number of processes)
- each process can be univocally identified
- processes are equals

Two processes can communicate iff they belong to the same communicator.

The default communicator is **MPI_COMM_WORLD**, which includes n processes (having rank = 0,..,n-1).

MPI_COMM_WORLD is a handle defined in mpi.h



Rank and size

To know its rank, a process has to call:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

To know the size of its communicator, a process has to call:

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Initializing and exiting MPI

The first MPI routine to be called is:

```
int MPI_Init(int *argc, char ***argv)
```

which initializes the default communicator (MPI_COMM_WORLD).

To exit from the MPI environment:

```
int MPI_Finalize()
```

Example: Hello, World!

Write a program that prints the rank of the process and the size of its communicator.

```
#include <stdio.h>
#include "mpi.h"

#define  MASTER 0

int main (int argc, char **argv)
{
  int   numtasks, taskid, len;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Get_processor_name(hostname, &len);
  printf ("Hello from task %d on %s!\n", taskid, hostname);

  if (taskid == MASTER)
     printf("MASTER: Number of MPI tasks is: %d\n", numtasks);

  MPI_Finalize();
}
```

Example: Hello, World!

Lesson learned:

- Compiling produces a unique executable

- Each command is executed by each process independently

- The runtime system (e.g., Open MPI) controls how the executable is replicated over computing nodes, how processes are created, how standard output/error are managed.

Point-to-Point Communication

MPI_Send

```
int MPI_Send(void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm)
```

where:
- buf is the pointer to the message to be sent (application buffer)
- count is the number of elements in the message
- datatype specifies the type of the elements in the message
- dest is the rank of the recipient
- tag is a non-negative integer whose purpose is left to the user
- comm is the communicator

MPI_Recv

```
int MPI_Recv(void *buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status *status)
```

where:
- buf is the pointer to the array where the received message must be stored
- count is the number of elements in the message
- datatype specifies the type of the elements in the message
- source is the rank of the sender
- only messages with the specified tag are considered for reception
- comm is the communicator
- status contains info about the envelope of the message to be received

Successful communication

A communication is successful if and only if:

- sender specifies a valid receiver's rank

- receiver specifies a valid source's rank

- sender and receiver are in the same communicator

- tags match

- datatypes correspond

- the receiver has a sufficiently large buffer

Wildcarding

It is possible to leave the sender unspecified (in the MPI_Recv), by means of MPI_ANY_SOURCE.

It is also possible to leave the tag unspecified, by means of MPI_ANY_TAG.

There is no wildcard for the communicator.

The status variable

The envelope of a message is given by the MPI_RECV struct, that can be retrieved from the status variable.

Other useful information are:

- Source: status.MPI_SOURCE
- Tag: status.MPI_TAG

Why retrieving source and tag from the status?
In case of receive with wildcard, it is the only way to know the sender and the type of the received message.

MPI_Get_count

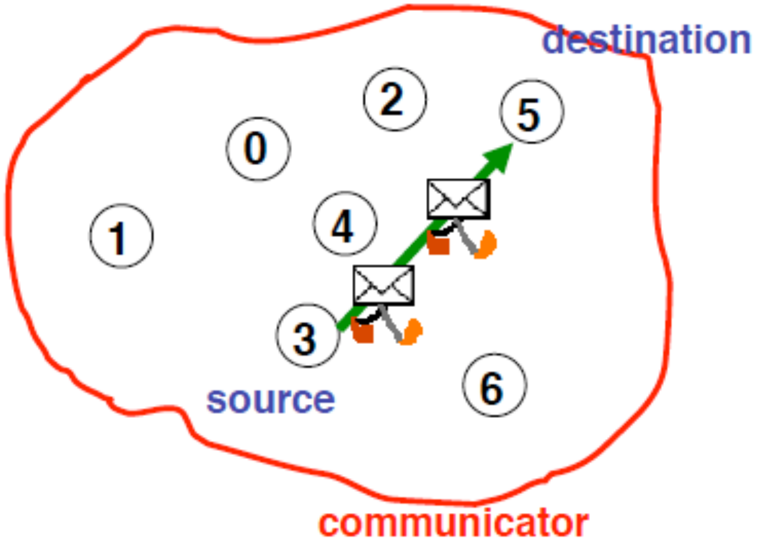
To get the number of elements with a specific datatype in the received message, the following function must be called on status:

```
int MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int *count)
```

Such a number is reported in count, after the function has been executed.

Message order preservation

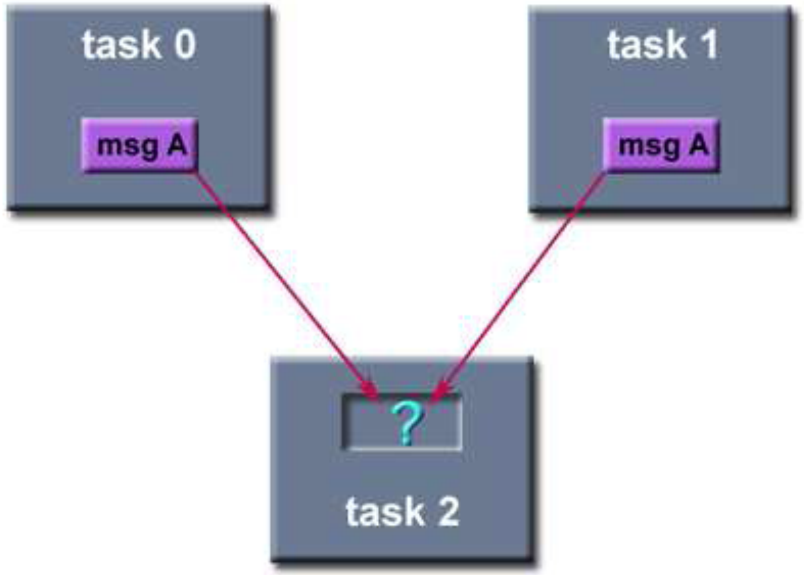
Messages from the same source, in the same communicator, with the same tag, to the same receiver are delivered in the same order they have been sent.



Avoiding starvation

The programmer must avoid starvation!

Process 0 and process 1 send the same message to process 2, that has set just one MPI_Recv. Thus, only one MPI_Send will succeed.



Example: send/receive an integer

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv)
{
        MPI_Status status;
        int rank, size;
        /* data to communicate */
        int data_int;
        /* initialize MPI */
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (rank == 0)
        {
                data_int = 10;
                MPI_Send(&data_int, 1, MPI_INT, 1, 666, MPI_COMM_WORLD);
        }
        else if (rank == 1)
        {
                MPI_Recv(&data_int, 1, MPI_INT, 0, 666, MPI_COMM_WORLD, &status);
                printf("Process 1 receives %d from process 0.\n", data_int);
        }
        MPI_Finalize();
        return 0;
}
```

Example: send/receive an array of floats

```
#include <stdio.h>
#include "mpi.h"
#define MSIZE 10

int main(int argc, char **argv) {
	MPI_Status status;
	int rank, size;
	int i, j;
	/*data to communicate */
	float a[MSIZE];

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	if (rank == 0) {
		for (i = 0; i<MSIZE; i++)
			a[i] = (float) i;
		MPI_Send(a, MSIZE, MPI_FLOAT, 1, 666, MPI_COMM_WORLD);
	}
	else if (rank == 1) {
		MPI_Recv(a, MSIZE, MPI_FLOAT, 0, 666, MPI_COMM_WORLD, &status);
		printf("Process 1 receives the following array from process 0.\n");
		for (i = 0; i<MSIZE; i++)
			printf("%6.2f\n", a[i]);
	}
	MPI_Finalize();
	return 0;
}
```

Example: send/receive part of an array of floats

```
#include <stdio.h>
#include "mpi.h"
#define VSIZE 50
#define BORDER 12

int main (int argc, char *argv[])
{
        MPI_Status status;
        int indx, rank, size, nprocs, i;
        int start_send_buf = BORDER;
        int start_recv_buf = VSIZE - BORDER;
        int length = 10;
        int vector[VSIZE];
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        /* all processes initialize vector */
        for (indx=0; indx<VSIZE; indx++)
                vector[indx] = indx;
        /* send length integers starting from the "start_send_buf"-th position of vector */
        if (rank == 0)
                MPI_Send(&vector[start_send_buf], length, MPI_INT, 1, 666, MPI_COMM_WORLD);
        /* receive length integers in the "start_recv_buf"-th position of vector */
        if (rank == 1) {
                MPI_Recv(&vector[start_recv_buf], length, MPI_INT, 0, 666, MPI_COMM_WORLD, &status);
                printf("Process 1 receives the following array from process 0.\n");
                for (i = 0; i<length; i++)
                        printf("%d\n", vector[start_recv_buf + i]);
        }
        MPI_Finalize();
        return 0;
}
```

Measuring execution time

```
double MPI_Wtime(void);
```

Standard timers (e.g., POSIX) are not adequate for MPI programs:
– insufficient accuracy
– not portable

The execution time of a task is measured by getting the current time before the execution starts, and after.

MPI_Wtime() returns the local time, measured from a prefixed time.

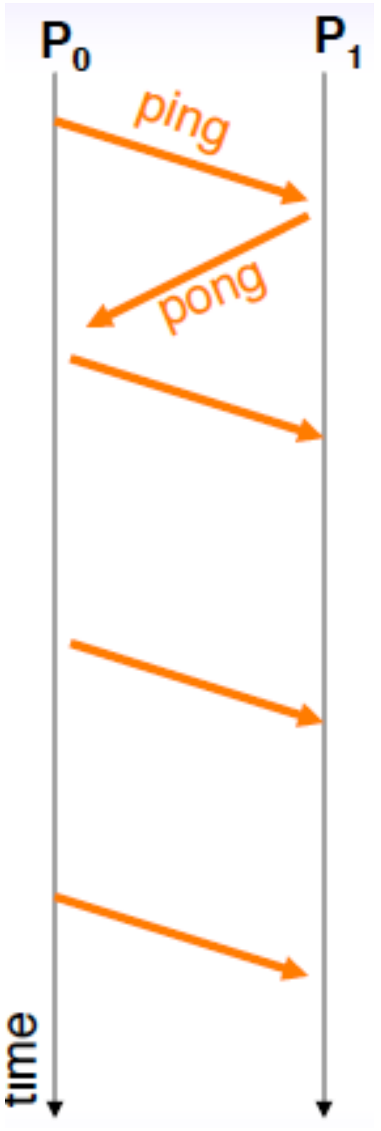
To know the resolution of the timer:

```
double MPI_Wtick(void);
```

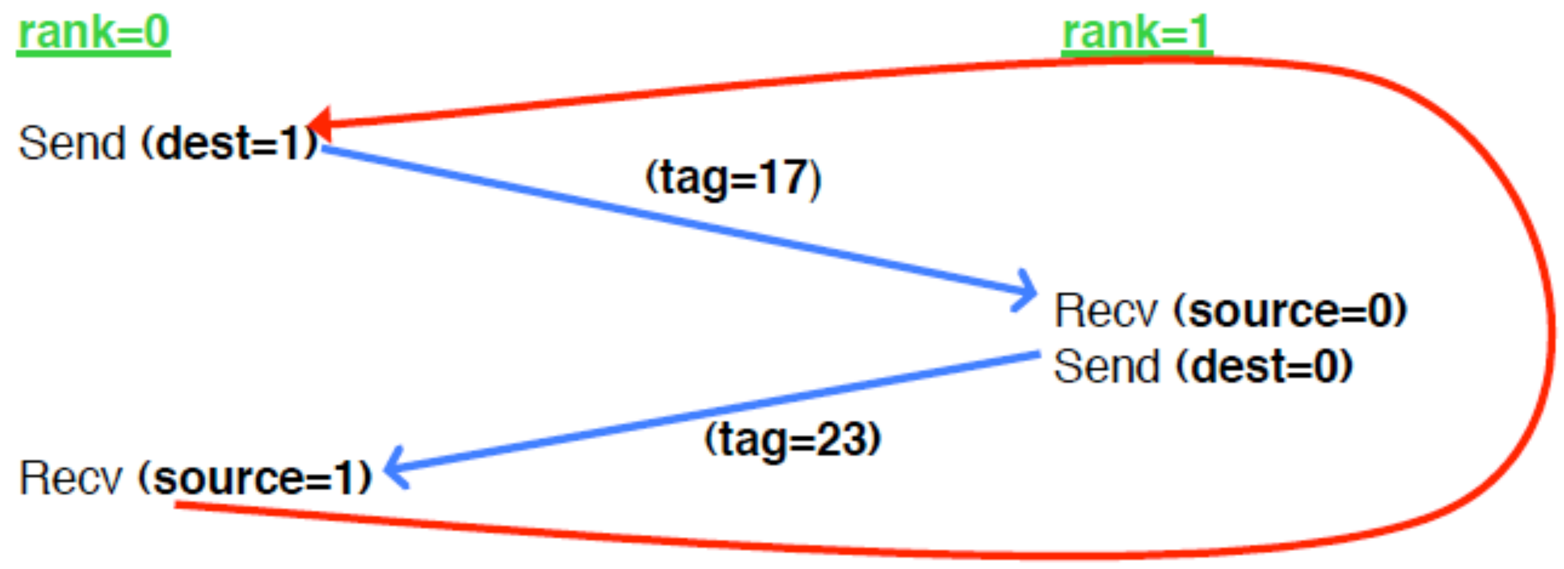
Example: PING-PONG

Write a program where two processes send to each other a message, according to the scheme in figure.

- process 0 sends the message to process 1 (PING)
- after receiving the message, process 1 sends it back to process 0 (PONG)
- etc. (repeat 50 times)



Example: PING-PONG



```
if (my_rank==0)                    /* i.e., emulated multiple program */
  MPI_Ssend( ... dest=1 ...)
  MPI_Recv( ... source=1 ...)
else
  MPI_Recv( ... source=0 ...)
  MPI_Ssend( ... dest=0 ...)
fi
```

Example: PING-PONG

```
#include <stdio.h>
#include <mpi.h>
#define proc_A 0
#define proc_B 1
#define ping 100
#define pong 101
#define number_of_messages 5
#define length_of_message 1

int main(int argc, char *argv[])
{
  int my_rank;
  float buffer[length_of_message];
  int i;
  double start, finish, time;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Example: PING-PONG

```
if (my_rank == proc_A)
{
  start = MPI_Wtime();
  for (i = 0; i < number_of_messages; i++)
  {
    buffer[0] = i;
    printf("Process %d sends %f to process %d.\n", proc_A, buffer[0], proc_B);
    MPI_Ssend(buffer, length_of_message, MPI_FLOAT, proc_B, ping, MPI_COMM_WORLD);
    MPI_Recv(buffer, length_of_message, MPI_FLOAT, proc_B, pong, MPI_COMM_WORLD, &status);
    printf("Process %d has received %f from process %d.\n", proc_A, buffer[0], proc_B);
  }
  finish = MPI_Wtime();
  time = finish - start;
  printf("Avg time for one message: %f seconds.\n", (float)(time / (2 * number_of_messages)));
}
```

Example: PING-PONG

```
 else if (my_rank == proc_B)
 {
   for (i = 0; i < number_of_messages; i++)
   {
     MPI_Recv(buffer, length_of_message, MPI_FLOAT, proc_A, ping, MPI_COMM_WORLD, &status);
     printf("Process %d has received %f from process %d.\n", proc_B, buffer[0], proc_A);
     printf("Process %d sends %f to process %d.\n", proc_B, buffer[0], proc_A);
     MPI_Ssend(buffer, length_of_message, MPI_FLOAT, proc_A, pong, MPI_COMM_WORLD);
   }
   printf("Final value for process %d is %f.\n", proc_B, buffer[0]);
 }

 MPI_Finalize();
}
```

Completed communications

A communication is **locally completed** on a process if the latter has completed its part of operations related to communication.

With respect to execution, local completion means that the process can execute the instruction that follows the SEND or RECV.

A communication is **globally completed** if all involved processes have completed their operations related to the communication.

A communication is globally completed if and only if it is locally completed on all involved processes.

Completed communications

The completion phase of the SEND function depends on the message size:
– Buffered for small size
– Synchronous for large size
In the first case the process can exit from the SEND after the message has been copied in a system buffer.
In the second case the process can exit from the SEND only when a RECV function is being executed.

Consider the following case:

```
if (myRank = 0)
 SEND A to Process 1
 RECV B from Process 1
else if (myRank = 1)
 SEND B to Process 0
 RECV A from Process 0
endif
```

There are two SEND that wait, and two RECV that can be executed only after the related SEND are completed.

Communication completion criteria

Synchronous send
➜ completed when the application buffer can be reused and the message reception has started

Buffered send
➜ completed when the message has been completely copied to the transfer buffer

Standard send
➜ completed when the application buffer can be reused

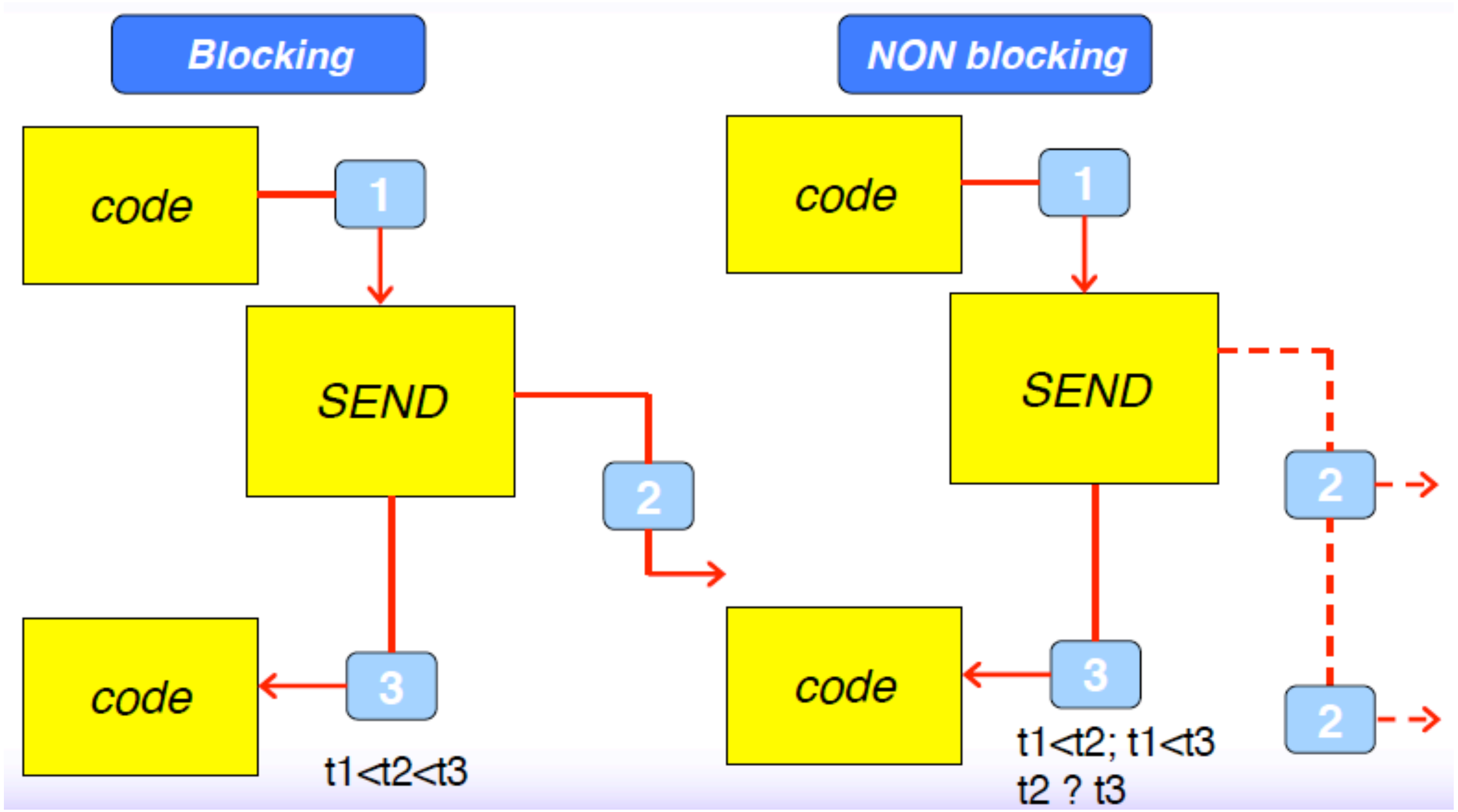
Receive
➜ completed when the message has arrived

Blocking and non blocking communication modes

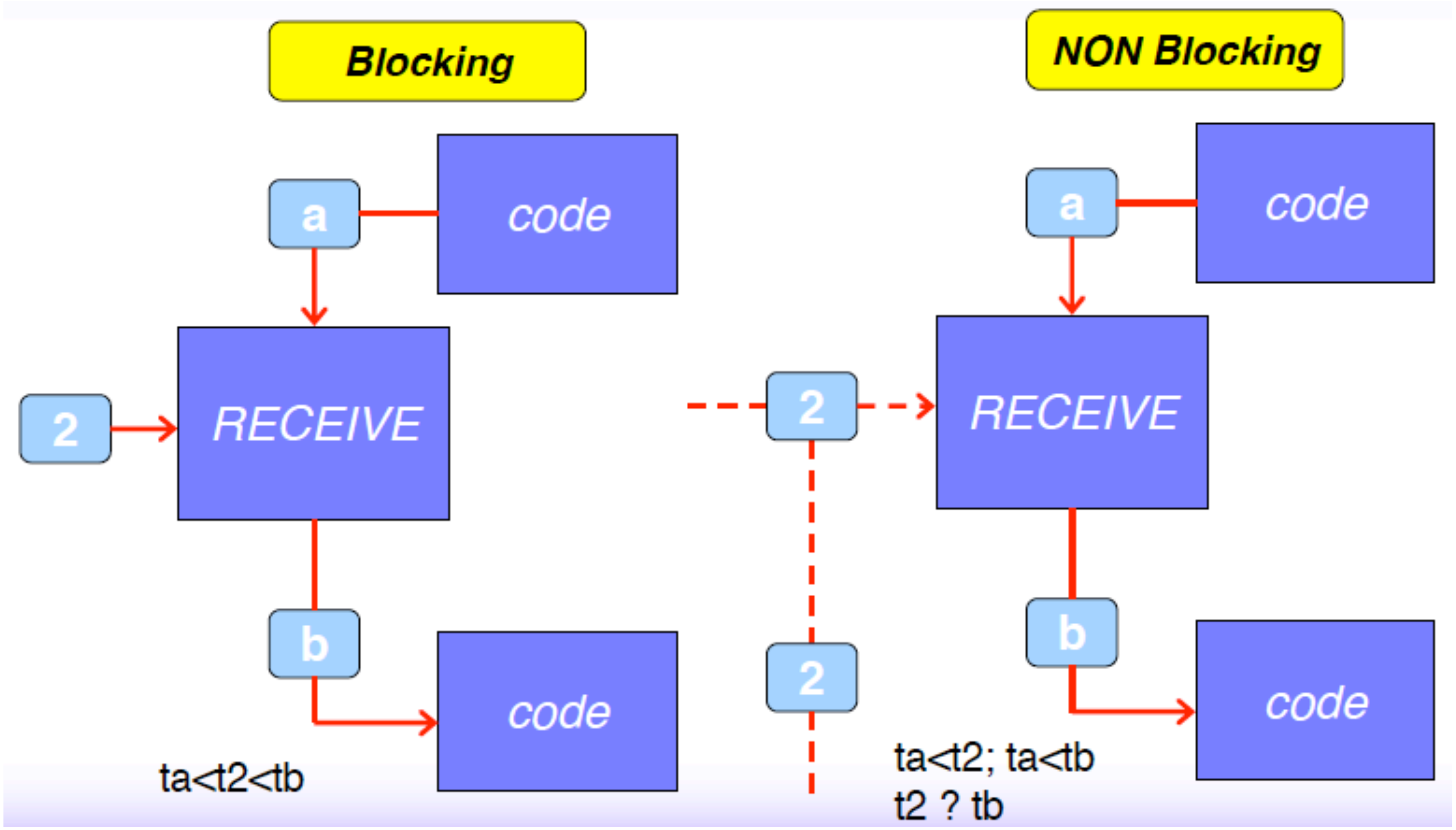
• **Blocking**:
– control is returned to the process that has invoked the communication primitive only when the latter has completed

• **Nonblocking**:
– control is immediately returned to the process that has invoked the communication primitive
– a check on the actual completion of the communication must be performed afterwards
– meanwhile, the process can perform other operations

Blocking and non blocking communication modes



Blocking and non blocking communication modes



Communication functions

Blocking:
- Synchronous send ➜ MPI_Ssend
- Buffered send ➜ MPI_Bsend
- Standard send ➜ MPI_Send
- Ready send ➜ MPI_Rsend
- Receive ➜ MPI_Recv

the I stands for “immediate”

Nonblocking:
- Synchronous non blocking send ➜ MPI_Issend
- Buffered non blocking send ➜ MPI_Ibsend
- Standard non blocking send ➜ MPI_Isend
- Ready non blocking send ➜ MPI_Irsend
- Non blocking receive ➜ MPI_Irecv

To check for completion ➜ MPI_Test, MPI_Wait

Synchronous send: MPI_Ssend

Sends the message and stays blocked until the application buffer of the sender is ready for being used for another operation **and** the receiver has started to receive the message.

Arguments are the same of the **MPI_Send**

Pros: This send is more secure than the standard one, because the network is not overloaded with pending messages. Sender and receiver are always synchronized (the resulting parallel program is more deterministic).

Cons: Risk of deadlocks. Risk of idle time.

Buffered send: MPI_Bsend

It is immediately completed as soon as the process has copied the message on a transfer buffer that must be explicitly managed by the program:

MPI_Buffer_attach(void *buf, int size)
allocates a memory area as transfer buffer

MPI_Buffer_detach(void *buf, int *size)
deallocates the memory area of the transfer buffer

Sender and receiver are not synchronized.

– **buf** is the memory address of the buffer
– **size** is an **int** (**INTEGER**) that defines the size **IN BYTES** of the buffer

Note: the transfer buffer is a user-managed alternative to the system buffer, which is MPI-managed.

Standard send: MPI_Send

Base blocking send operation: completed when the message has been sent and the application buffer can be reused.

The message can stay in the communication network for a while.

Ready send: MPI_Rsend

A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined.

It is better to avoid it, unless you are 200% sure that the related receive has been set.

Send-receive: MPI_Sendrecv

Executes a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype s_dtype, int dest, int stag, void *dbuf, int dcount, MPI_Datatype d_type, int src, int dtag, MPI_Comm comm, MPI_Status *status)

The first half of the list of arguments is related to the send, the second half to the receive.

– [IN] **dest** is the *rank* of the *receiver* within the communicator **comm**
– [IN] **stag** is the *identifier* of the *sent message*
– [IN] **src** is the *rank* of the *sender* within the communicator **comm**
– [IN] **dtag** is the *identifier* of the *received message*

Nonblocking communications

A nonblocking communication is tipically made by three phases:
1. the beginning of the send/receive of the message;
2. the execution of an activity that does not imply the access to the data involved in the communication;
3. the wait for the completion of the communication;

Pros:
Performance: a nonblocking communication allows one to:
– overlap communication phases with computing phases;
– reduce the effects of the communication latency.
Non blocking communications avoid deadlock situations.

Cons:
Programming nonblocking message passing operations is more complicate.

Synchronous nonblocking send

MPI_Issend(buf, count, datatype, dest, tag, comm, &request_handle)

MPI_Wait(&request_handle, &status)

- buf cannot be used between Issend and Wait
- Issend immediately followed by Wait is equivalent to Ssend
- status is not used in the Issend, but in the Wait
- request_handle allows the Wait to refer to a precise send
- wait and test allow to know when the receiver has received the message

NOTE: “I” stands for “immediate” (because the function returns the control to the process immediately)

Standard nonblocking send

MPI_Isend(void *buf, int count, MPI_Datatype datatype,int dest,int tag, MPI_Comm comm, MPI_Request *request)

- Initialize the send operation

- Identifies a memory area that must be used as a buffer; the execution goes on without waiting that the message is being copied by the application buffer

- a request handle is returned to manage the state of the pending message

- a program cannot modify the application buffer until a subsequent call to MPI_Wait or MPI_Test indicate that the send has completed

- may be related to a blocking receive

- improves the overlapping degree between computing and communication

Nonblocking receive

int MPI_Irecv(void buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request request)

- Initializes the receive operation

- Identifies a memory area as receive buffer

- The function returns immediately, not waiting for the message to be fully copied in the local application buffer

- buf cannot be used while the request is pending (use MPI_Wait or MPI_Test to know when the request has been fulfilled)

- to check for the status of the receive request, use the request handle that is returned (the blocking MPI_Recv has the status variable, instead)

Completion test

When using point-to-point nonblocking communications, it is fundamental to check for the completion of the communication, before
- using the receive buffer
- using the send buffer

MPI provides two types of check operations:
- *WAIT* allows to stop the execution of the process until the communication completes
- *TEST* returns to the calling process a value that is TRUE if the communication has completed, FALSE otherwise

MPI_Wait(MPI_request request, MPI_Status status)

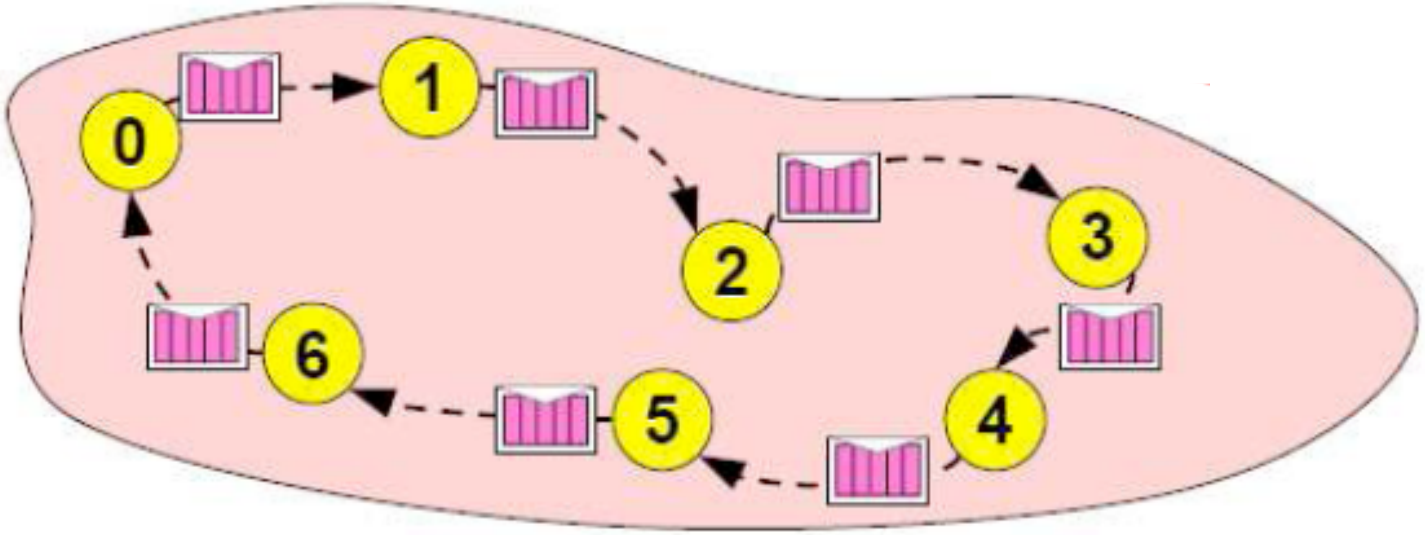
MPI_Test(MPI_Request request, int flag, MPI_Status status)

Deadlocks

Consider the following code:

```
MPI_Ssend(…, right_rank, …)
MPI_Recv(…, left_rank, …)
```

The Ssend does not complete until the corresponding Recv has been started, but since all processes are blocked on SSend, no one starts the Recv.



Deadlocks

How to avoid deadlocks:

1) change the order of the calls
- dangerous if dependencies involve more than 2 processes
- require clear knowledge of the communication pattern (it must be deterministic)
- does not add complexity to the program

2) use non blocking operations
- adds complexity to the program
- may improve efficiency

3) use buffered modes
- buffer must be carefully managed
- does not add too much complexity to the program
- adds determinism to the program

Multiple non blocking communications

It is possible that many non blocking communications are posted at the same time, for which MPI provides operations for simultanously checking their completion:

a) Wait or test for completing **one** message in a list:
MPI_Waitany / MPI_Testany

b) Wait or test for completing **all** messages:
MPI_Waitall / MPI_Testall

c) Wait or test for completing **some** messages:
MPI_Waitsome / MPI_Testsome

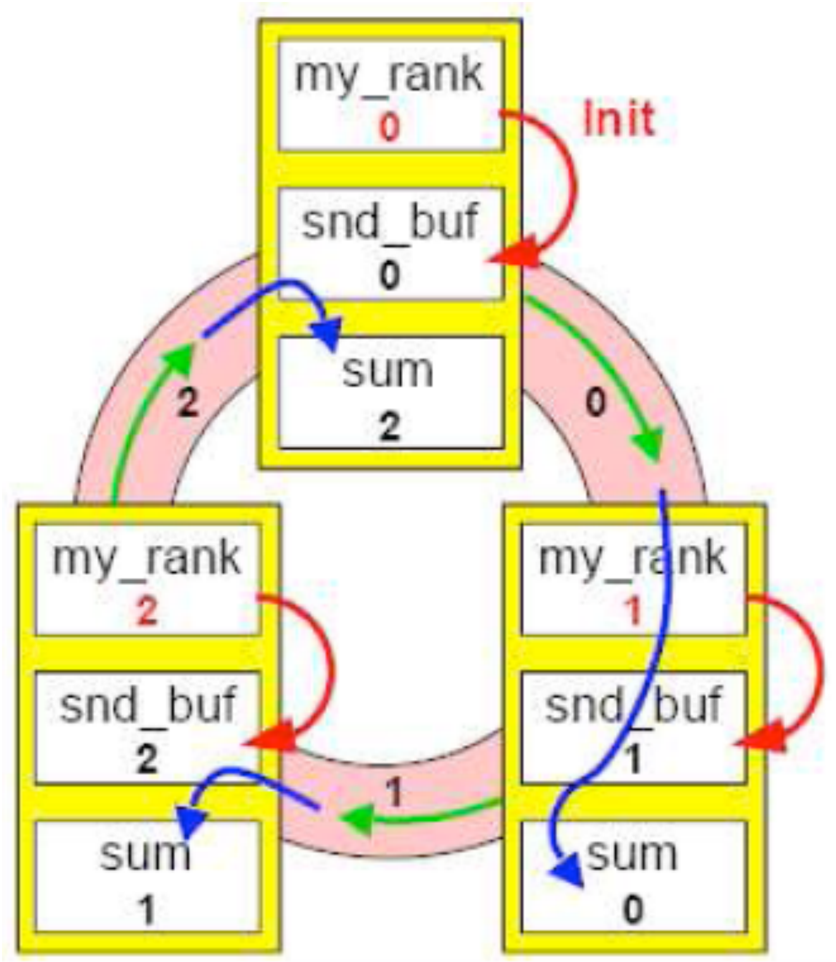
Example - propagate message on a ring

Consider a set of N processes organized as a ring.

- Initially, every process saves its MPI_COMM_WORLD rank in a variable called snd_buf.

- Then, every process repeats N times:
• send the snd_buf value to the neighbor on the right;
• receive a value, add it to the sum variable, then copy it to snd_buf

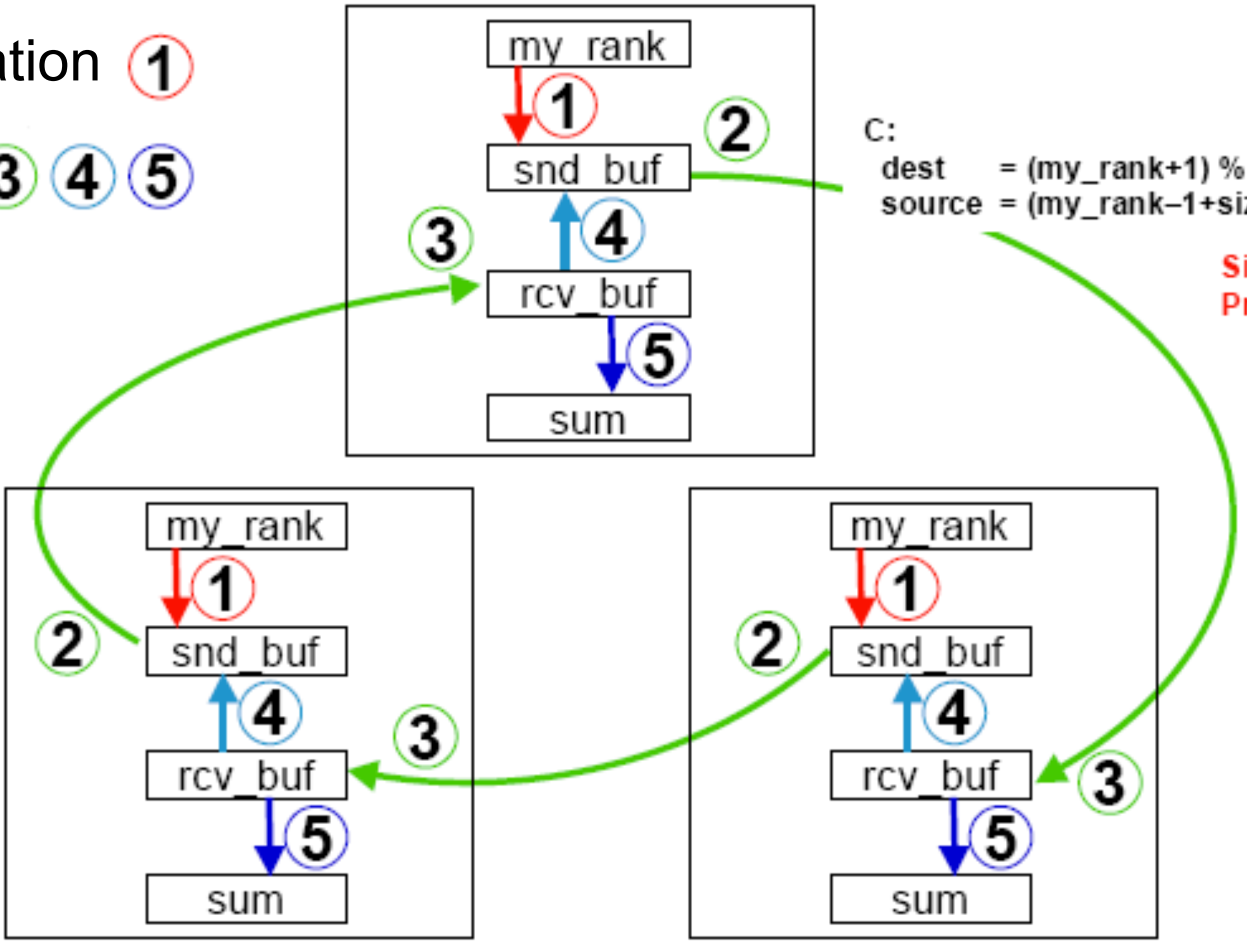
• MPI_Issend (nonblocking) is used
– to avoid deadlocks;
– to check the correctness, because the syncronous send would cause a deadlock.



Example - propagate message on a ring

Initialization ①

Iterations ② ③ ④ ⑤



```
C:
  dest    = (my_rank+1) % size;
  source  = (my_rank–1+size) % size;
```

Single Program !!!

Example - propagate message on a ring

```
#include <stdio.h>
#include <mpi.h>
#define to_right 201

int main (int argc, char *argv[])
{
  int my_rank, size;
  int snd_buf, recv_buf;
  int right, left;
  int sum, i;
  MPI_Status status;
  MPI_Request request;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  right = (my_rank+1) % size;
  left = (my_rank-1+size) % size;
```

Example - propagate message on a ring

```
/* ... this SPMD-style neighbor computation with modulo has the same meaning as: */
/* right = my_rank + 1; */
/* if (right == size) right = 0; */
/* left = my_rank - 1; */
/* if (left == -1) left = size-1;*/
sum = 0;
snd_buf = my_rank;
for( i = 0; i < size; i++)
{
  MPI_Issend(&snd_buf, 1, MPI_INT, right, to_right, MPI_COMM_WORLD, &request);
  MPI_Recv(&recv_buf, 1, MPI_INT, left, to_right, MPI_COMM_WORLD, &status);
  MPI_Wait(&request, &status);
  printf("Process %d:\tReceived: %d\n", my_rank, recv_buf);
  sum += recv_buf;
  snd_buf = recv_buf;
}
printf ("Process %d:\tSum = %d\n", my_rank, sum);
MPI_Finalize();
}
```

Derived Datatypes

Motivation

Derived datatypes are used for **polymorphic messages**, i.e., messages that are made of compound items comprising various generic datatypes, e.g., integers, floats, and characters (all in a single data block).

Derived datatypes are also used to transfer data from non-contiguous buffers, e.g., a portion of a matrix.

To this effect MPI lets programmers specify mixed and non-contiguous communication buffers, so that objects of various shapes and sizes can be transferred directly without copying.

Motivation

Problem: to specify non-contiguous data of a single type, or contiguous data of mixed types, or non-contiguous data of mixed types.

A few possible solutions are that you could:

- make multiple MPI calls to send and receive each data element

- use MPI_Pack to combine different datatypes into contiguous memory for sending and MPI_Unpack to unpack the data back into non-contiguous memory after being received (datatype MPI_PACKED)

- **use MPI_BYTE to get around the datatype-matching rules**. MPI_BYTE can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte.

Generally, however, these solutions are slow, clumsy, and wasteful of memory. Using MPI_BYTE or MPI_PACKED might also result in a program that isn't portable to a heterogeneous system of machines. The idea of MPI derived datatypes is to provide a portable and efficient way of communicating non-contiguous or mixed types in a message. MPI derived datatypes provide a simpler, cleaner, more elegant and efficient way to handle this type of data.

Creating derived datatypes

Derived datatypes are created at run-time before being used in a communication, by means of the appropriate constructor, which populates a datatype descriptor (whose type is MPI_Datatype):

- int MPI_Type_contiguous(.., MPI_Datatype *newtype)
- int MPI_Type_vector(.., MPI_Datatype *newtype)
- int MPI_Type_struct(.., MPI_Datatype *newtype)

A created datatype must be committed, using
int MPI_Type_commit(MPI_Datatype *newtype)

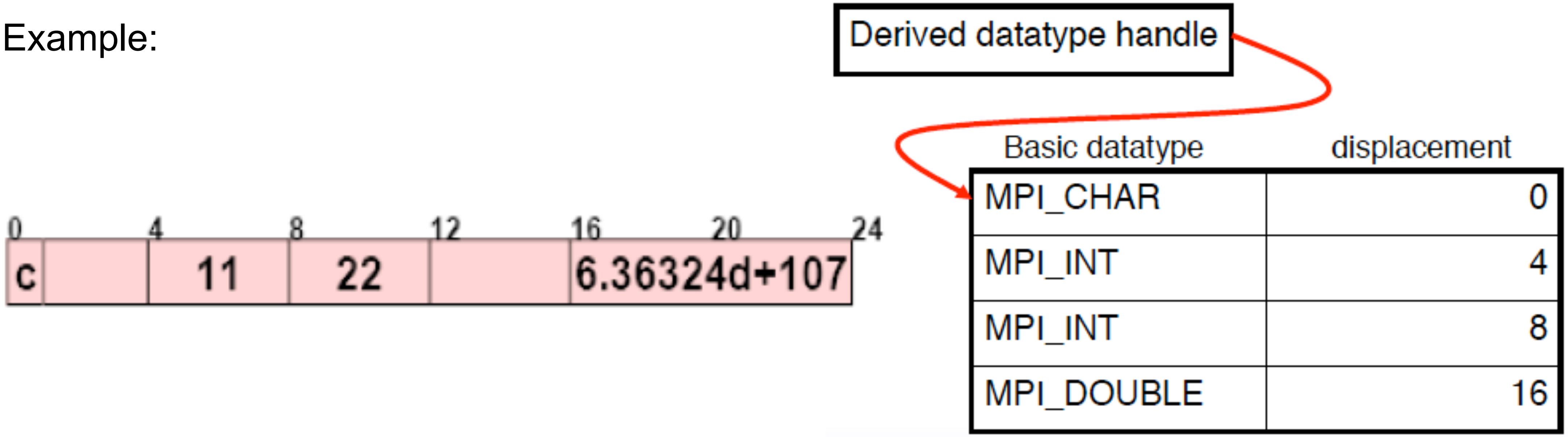
After being used, derived datatypes mus be deallocated, using
int MPI_Type_free(MPI_Datatype *newtype)

Type map

Type map of a derived datatype:

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

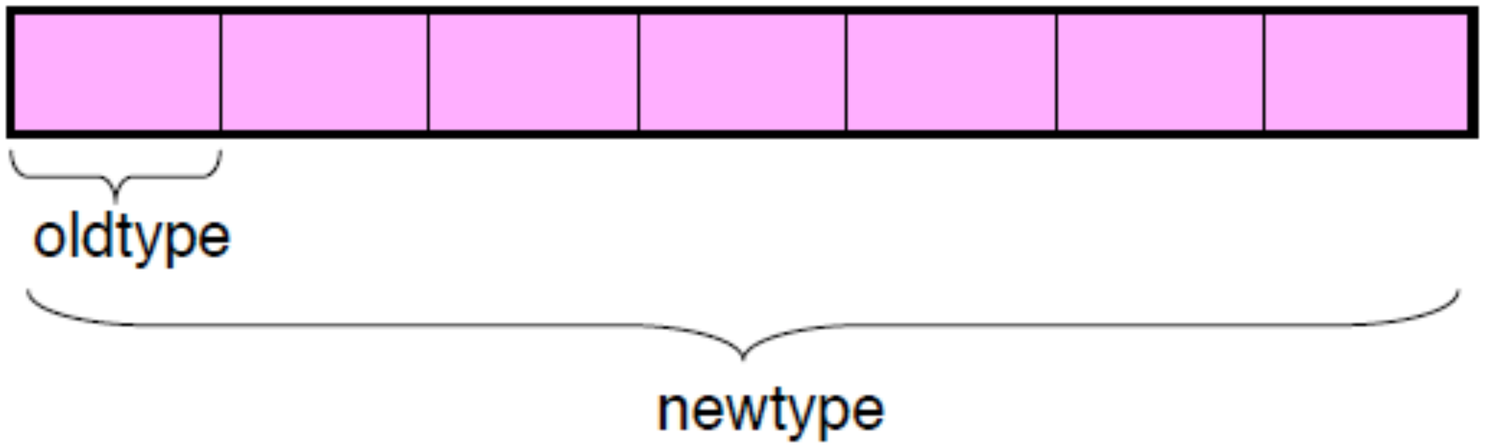
Example:



Basic datatype	displacement
MPI_CHAR	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16

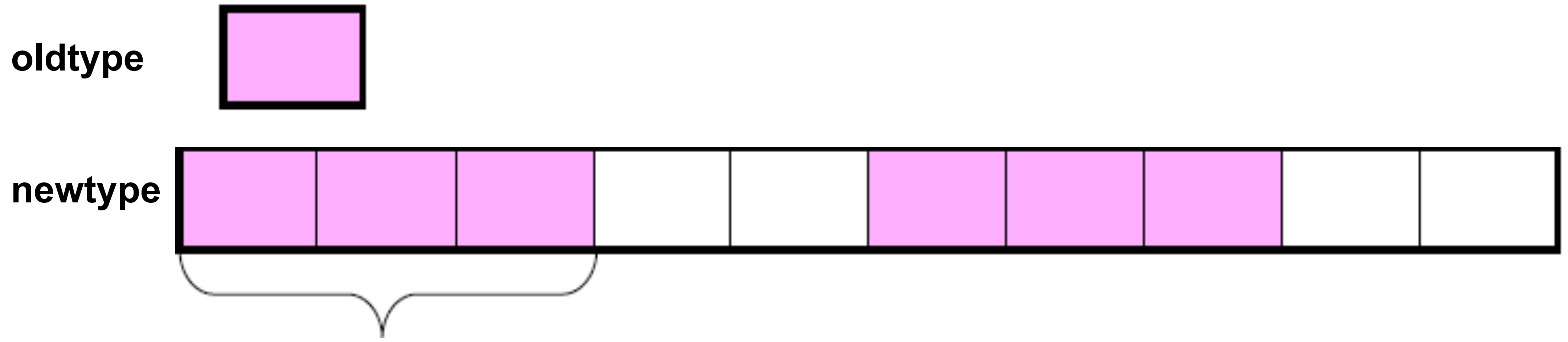
MPI_Type_contiguous

The most simple derived datatype is a set of count elements of the same type:



int MPI_Type_contiguous(int count, MPI_Datatype *oldtype, MPI_Datatype *_newtype_)

MPI_Type_vector



blocklength = 3 elements per block

stride = 5 (step between two blocks of elements)

count = 2 (blocks)

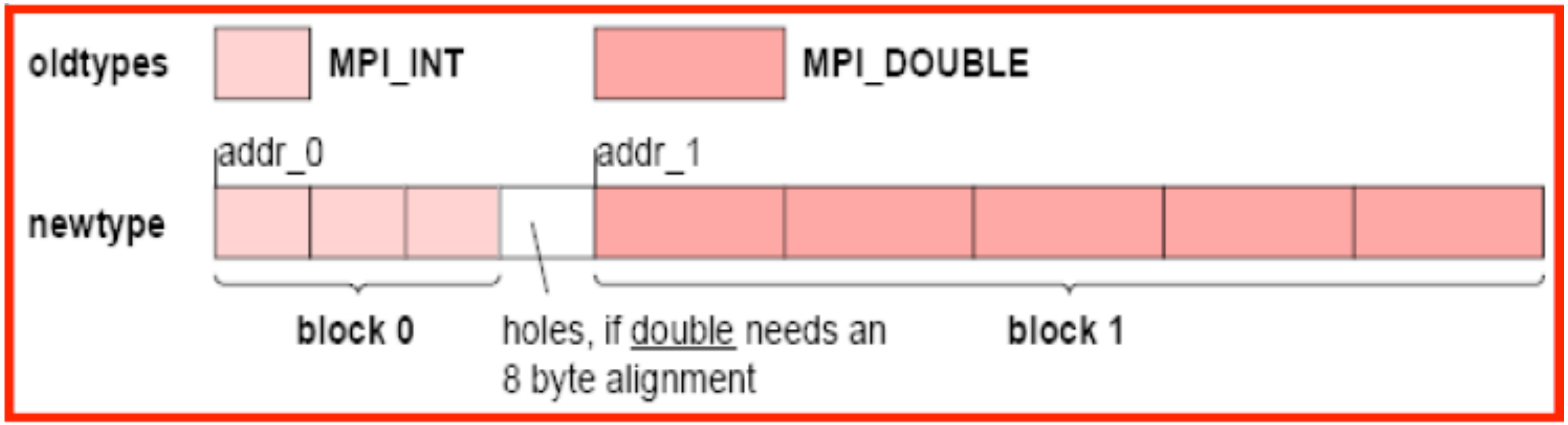
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype *oldtype, MPI_Datatype *newtype)

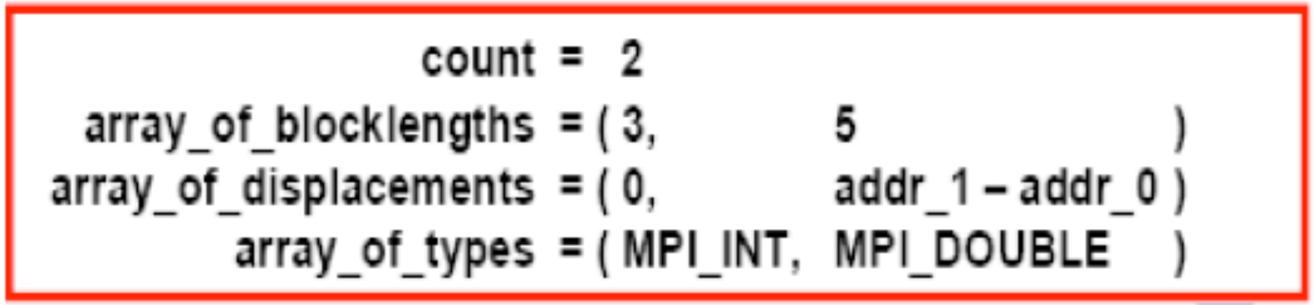
Type struct

```
int MPI_Type_create_struct(int count,
    const int *array_of_blocklengths,
    const MPI_Aint *array_of_displacements,
    const MPI_Datatype *array_of_types,
    MPI_Datatype *newtype)
```

count: number of blocks (integer) - also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
array_of_blocklengths: number of elements in each block (array)
array_of_displacements: byte displacement of each block (array)
array_of_types: type of elements in each block (array of handles to datatype objects)
newtype: new datatype (handle)

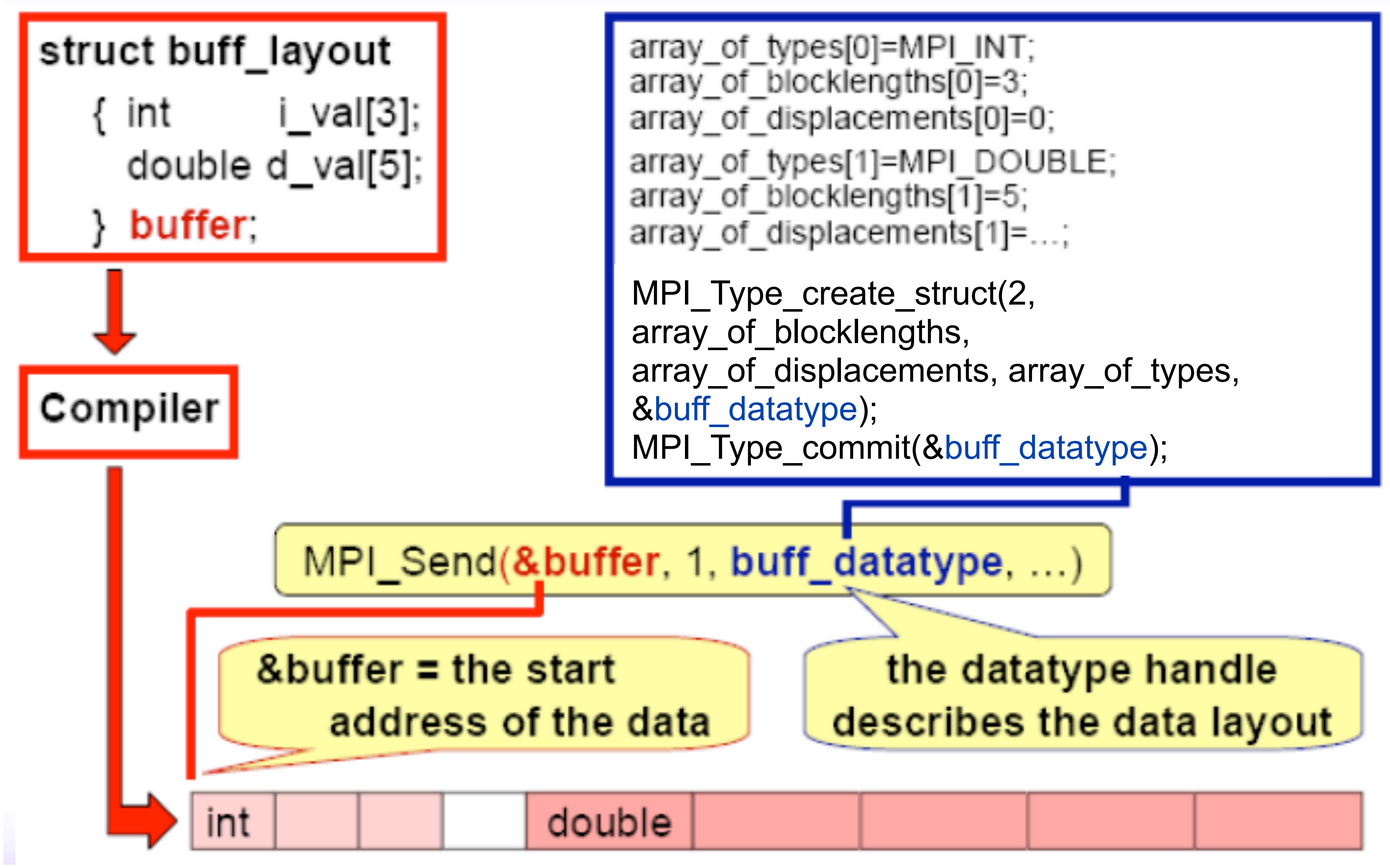
Type struct example





```
                count  =   2
   array_of_blocklengths  = ( 3,          5               )
  array_of_displacements  = ( 0,          addr_1 – addr_0 )
          array_of_types  = ( MPI_INT,   MPI_DOUBLE      )
```

Data layout and buffer



```
struct buff_layout
    { int         i_val[3];
      double d_val[5];
    } buffer;
```

```
array_of_types[0]=MPI_INT;
array_of_blocklengths[0]=3;
array_of_displacements[0]=0;
array_of_types[1]=MPI_DOUBLE;
array_of_blocklengths[1]=5;
array_of_displacements[1]=…;

MPI_Type_create_struct(2,
array_of_blocklengths,
array_of_displacements, array_of_types,
&buff_datatype);
MPI_Type_commit(&buff_datatype);
```

```
MPI_Send(&buffer, 1, buff_datatype, …)
```

&buffer = the start address of the data
the datatype handle describes the data layout
int
double

Memory layout

Fixed memory layout of the struct datatype in the previous example:

```
struct buff_layout
{
  int i_val[3];
  double d_val[5];
}
```

Arbitrary memory layout:

- Each array is allocated independently
- Each buffer is a copy of a 3-int-array and of a 5-double-array
- For each buffer it is necessary a specific datatype handle

Computing a displacement

array_of_displacements[i] := address(block_i) – address(block_0)

where address (of a location in memory) is obtained by means of

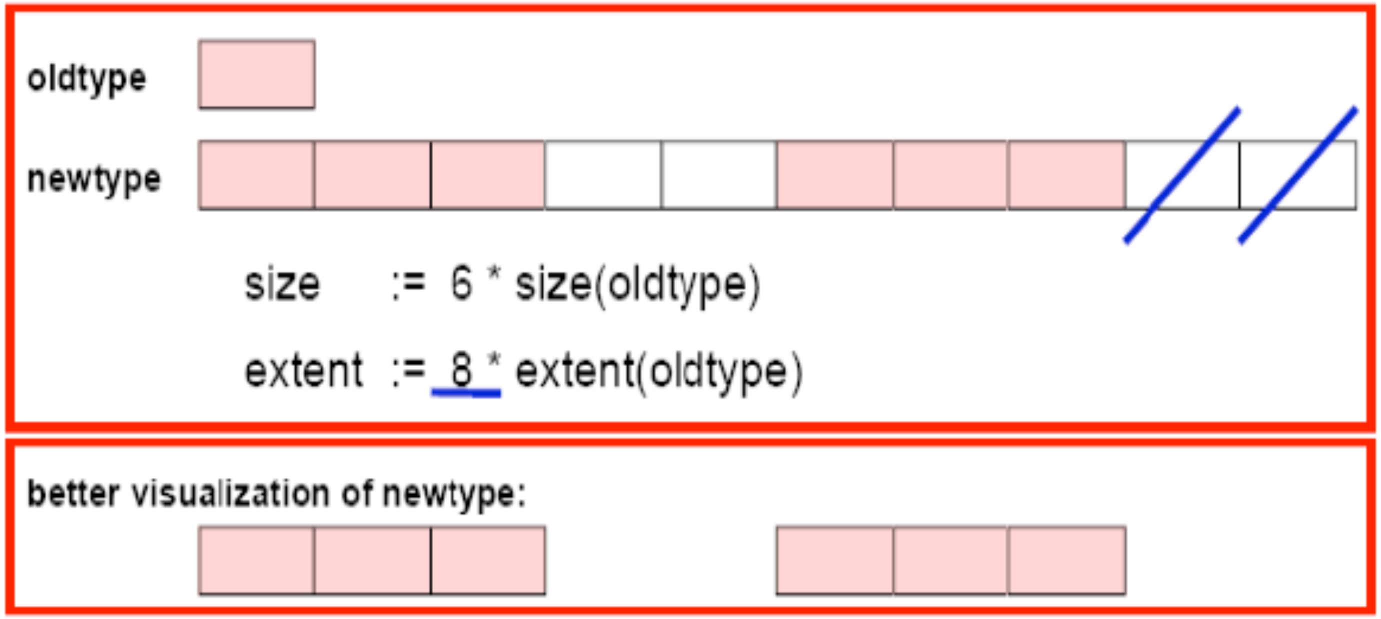
int MPI_Get_address(void *location, MPI_Aint *address)

Size and extent of a datatype

Size = number of bytes that must be transferred
Extent = interval between the first and the last byte

For base types: Size = Extent = number of bytes used by the compiler

For derived datatypes:


oldtype
newtype
size := 6 * size(oldtype)
extent := 8 * extent(oldtype)
better visualization of newtype:

Size and extent of a datatype

int MPI_Type_size(MPI_Datatype datatype, int *size)

int MPI_Type_get_extent (MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)

Example of contiguous data

- create a datatype that represents the row of a matrix
- send a different row to each process

```
#include <stdio.h>
#include <mpi.h>
#define SIZE 4

int main (int argc, char *argv[])
{
  int numtasks, rank, source=0, dest, tag=1, i;
  float a[SIZE][SIZE] = {1.0, 2.0, 3.0, 4.0, // matrix
  5.0, 6.0, 7.0, 8.0,
  9.0, 10.0, 11.0, 12.0,
  13.0, 14.0, 15.0, 16.0};
  float b[SIZE]; // array
  MPI_Status stat;
  MPI_Datatype rowtype;
  MPI_Init(&argc,&argv);
  ...
```

Example of contiguous data

```
  ...
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
  MPI_Type_commit(&rowtype);
  if (numtasks == SIZE) {
    if (rank == 0) {    // master
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
    }
    MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat); // source is 0 (master)
    printf("rank = %d, b = %3.1f %3.1f %3.1f %3.1f\n", rank,b[0],b[1],b[2],b[3]);
  }
  else
    printf("Must specify %d processors. Terminating.\n", SIZE);
  MPI_Type_free(&rowtype);
  MPI_Finalize();
}
```

Collective Communication

Motivation

MPI provides functions that implement communication patterns involving more processes, thus avoiding the programmer to implement them by combining point-to-point communications (which is ennoying and error-prone).

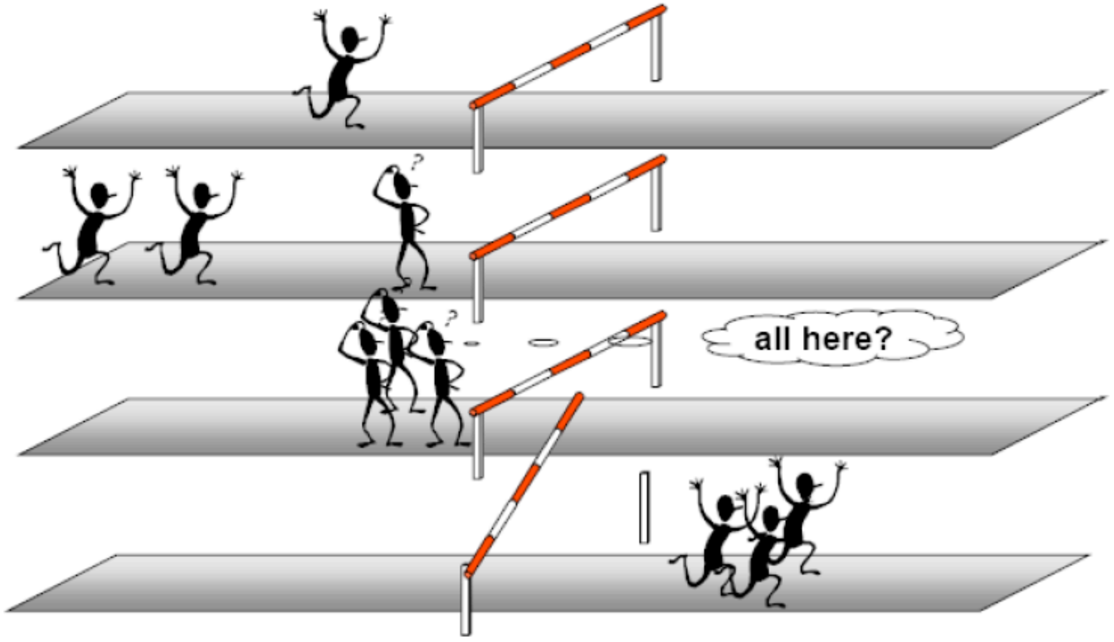
Three classes: ***all-to-one***, ***one-to-all*** and ***all-to-all***.

Collective operations:
- all processes must communicate, i.e., call the collective routine with same arguments
- blocking and nonblocking (MPI 3.0)
- no tags
- receive buffers must fit exactly the message size

Barrier synchronization

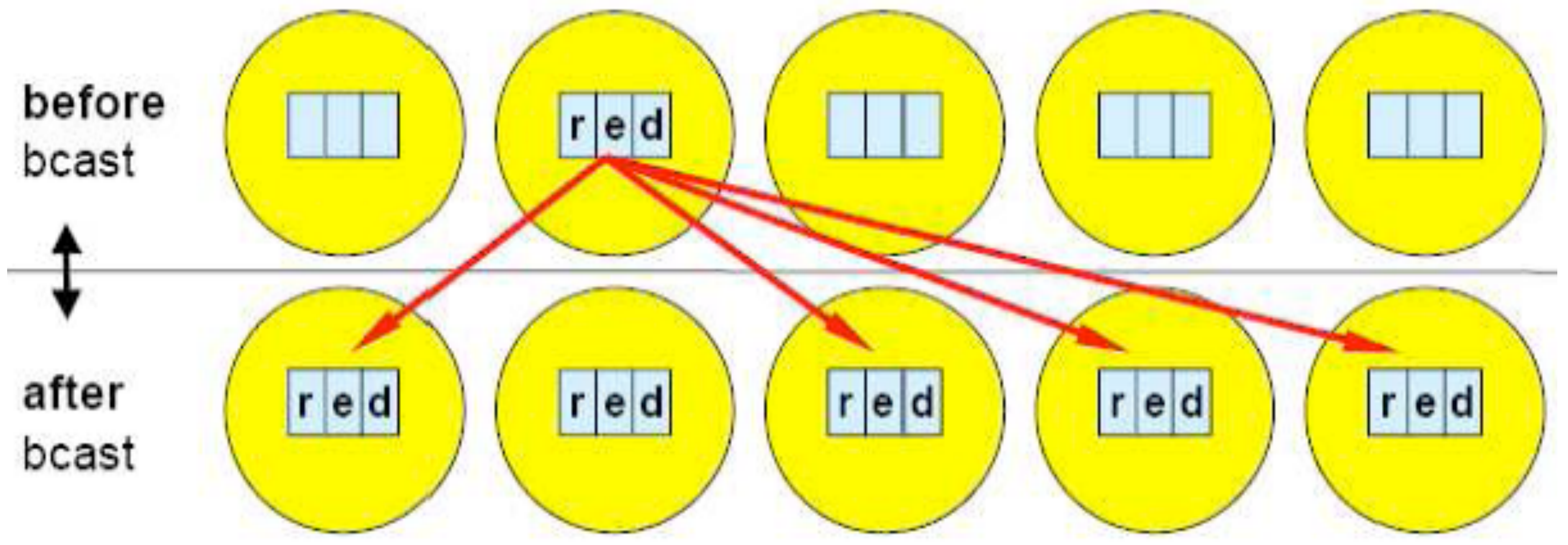
int MPI_Barrier(MPI_Comm comm)

Usually it is not necessary, because processes already synchronize with communication (a process cannot advance without all needed data).



Broadcast

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)



Broadcast

- replicates the content of the root’s buffer to all other processes within the communicator
- does not imply synchronization, necessarily

Parameters:

• **buffer** (in/out) address of the send/receive buffer
• **count** (in) number of elements forming the data to be sent
• **datatype** (in) type of the data to be sent
• **root** (in) rank of the sender
• **comm** (in) name of the comunicator

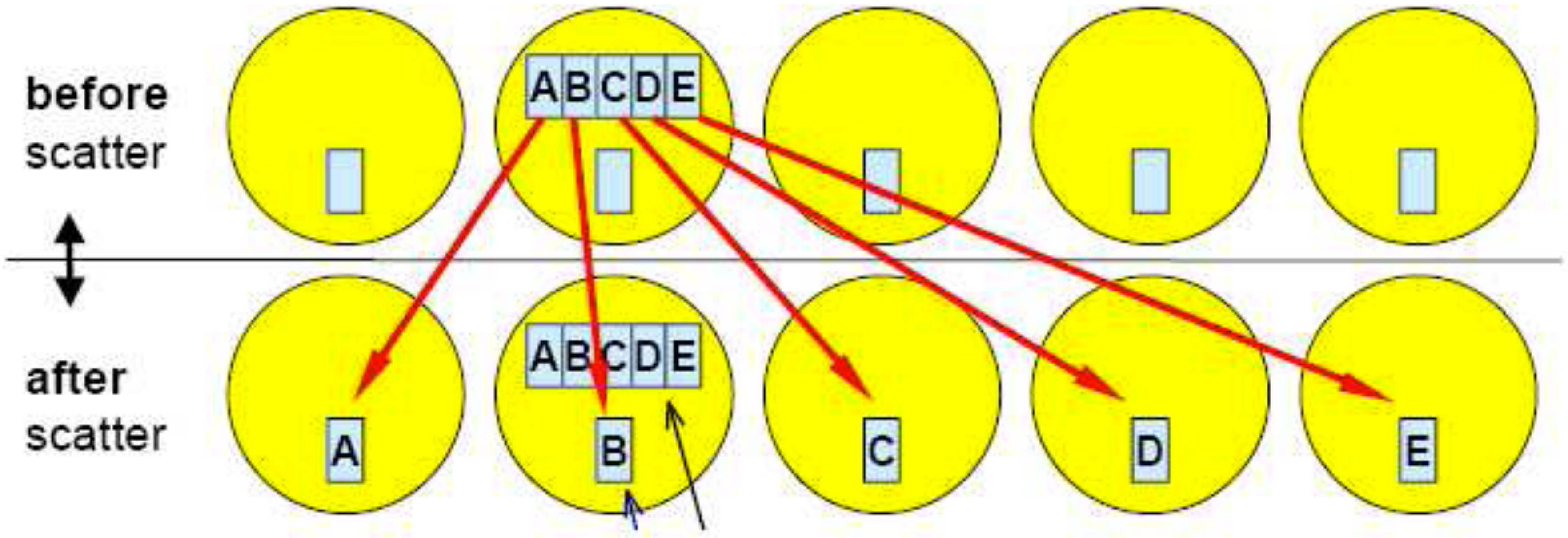
Broadcast belongs to the one-to-all class.

Broadcast

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
        int rank;
        int buf;
        const int root = 0;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if (rank == root) {
           buf = 777;
        }

        printf("[%d]: Before Bcast, buf is %d\n", rank, buf);

        /* everyone calls bcast, data is taken from root and ends up in everyone's buf */
        MPI_Bcast(&buf, 1, MPI_INT, root, MPI_COMM_WORLD);

        printf("[%d]: After Bcast, buf is %d\n", rank, buf);

        MPI_Finalize();
        return 0;
}
```

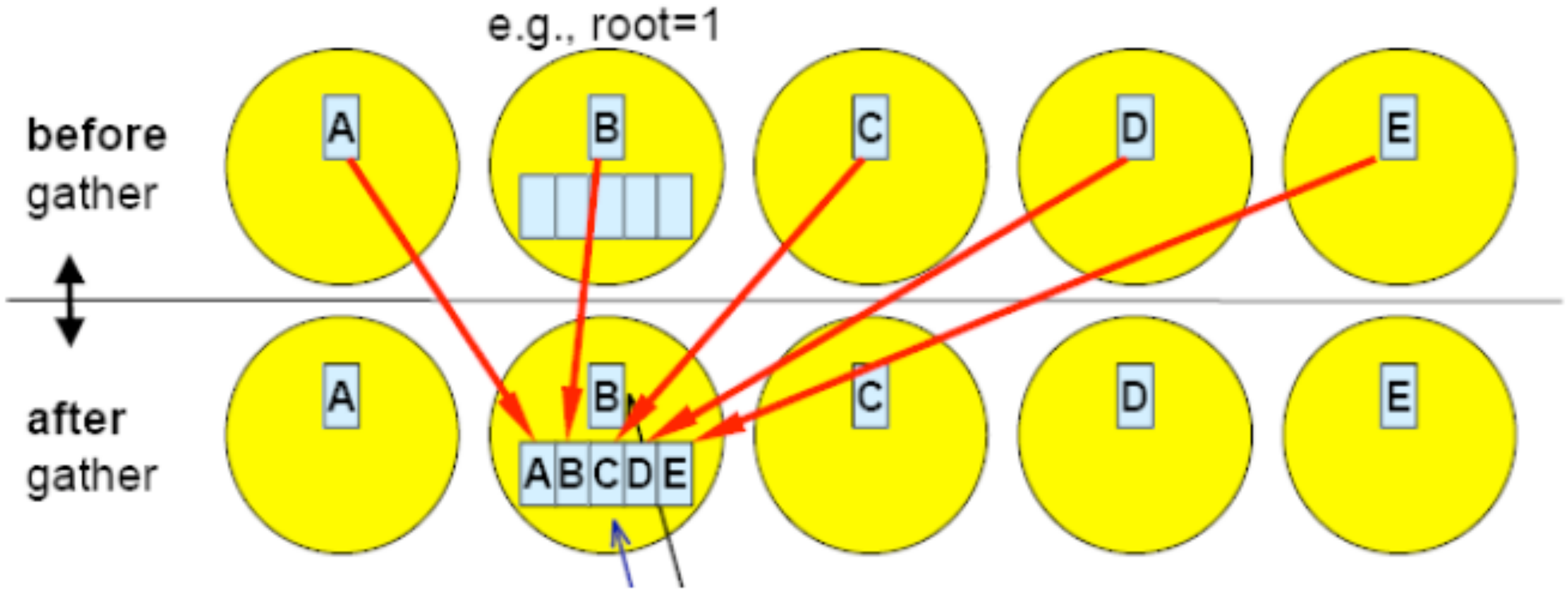
Scatter

int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)



Gather

int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)



Gather

– Every process (included the root one) sends the content of its send buffer to the root process
– The root process receives the data and order them based on the rank of the sender

Gather belongs to the “all-to-one” class.

There is also MPI_All_gather, which is equivalent to a MPI_Gather operation followed by a broadcast executed by the root process.

Reduction

used to perform computations that involve distributed data within a group of processes.

Examples:
- global sum and product
- global max and min
- user-defined global operations

Suppose to have an **associative and commutative operator o** that, given two elements of the same datatype, produces a result of the same type:

$d_0 \circ d_1 \circ d_2 \circ d_3 \circ \ldots \circ d_{s-2} \circ d_{s-1}$

d_i = data in process with rank **i** (single variable or vector)

Reduction

int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Example:

```
root = 0;
MPI_Reduce(&localvalue, &globalvalue, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

The sum of all localvalues (each one being an integer) is returned in globalvalue of process with rank 0.

Reduction

MPI Name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or (xor)
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise xor
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Reduction

User-defined reduction operators must be implemented as:

```
for (i = 1 to len)
    inoutvec[i] = inoutvec[i] OP invec[i]
```

where OP is associative.

Function signature:

void my_operator(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype)

Reduction

What happens, actually?

Suppose that two tasks have sendbuffers that contain two floats each:

sendbuf1[] = {f11, f12}, sendbuf2 = {f21,f22}

and perform MPI_MAX with reduction (with task 1 playing the root). The result is:

recvbuf[] = {max(f11,f21), max(f12,f22)}

What happened? This:

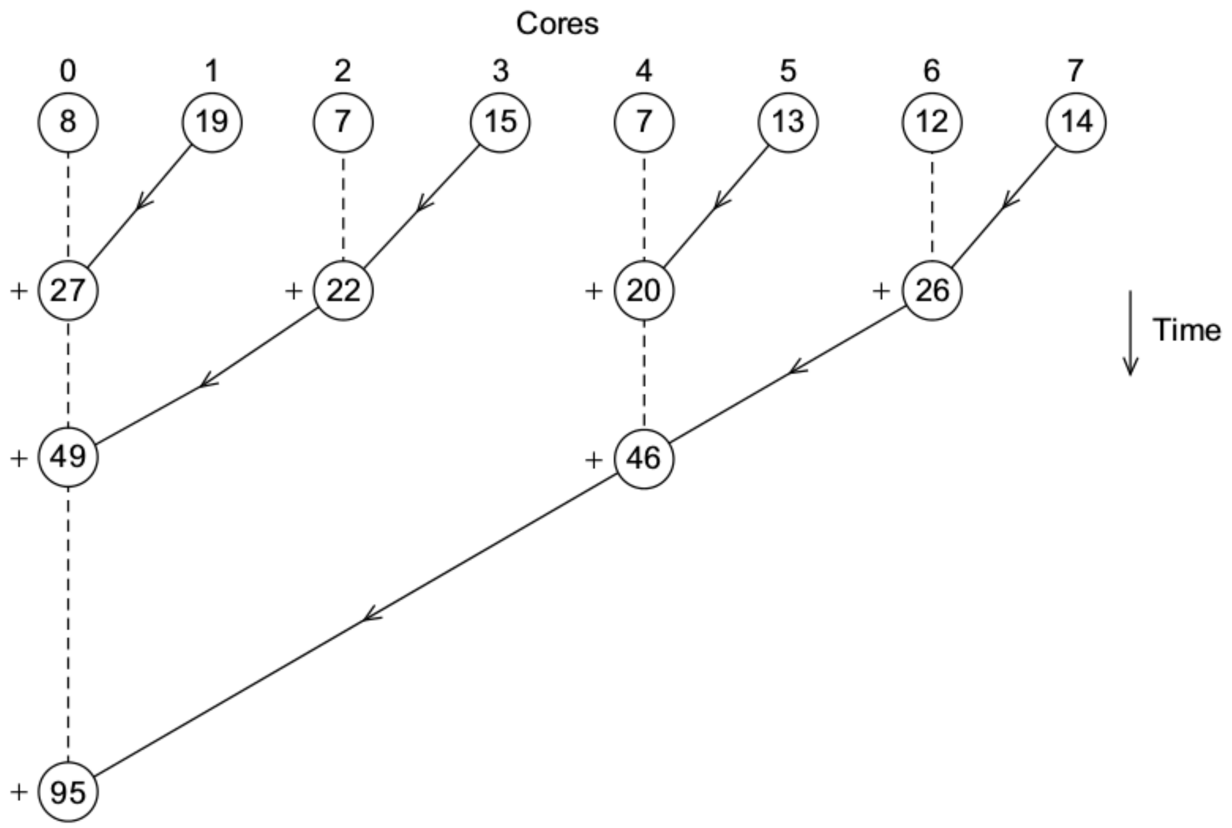
recvbuf[0] ← sendbuf2[0] (communication)
recvbuf[0] = recvbuf[0] OP sendbuf1[0]
recvbuf[1] ← sendbuf2[1] (communication)
recvbuf[1] = recvbuf[1] OP sendbuf1[1]

Which means that recvbuf is the inoutvec, while sendbuf1 is the invec.

Reduction

The scheme can be generalized to any associative OP.

And if there are more than 2 tasks, MPI builds a binary tree of communicating tasks:



Reduction

To register an user-defined operation:

int MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)

to delete the operation:

int MPI_Op_Free(op)

Reduction

Example: 1-norm

$$\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|$$

where x is a vector of size n.

Step 1: Implement your own reduction operator.

```
void onenorm(float *in, float *inout, int *len, MPI_Datatype *type) {
 int i;
   for (i=0; i<*len; i++) {
    *inout = fabs(*in) + fabs(*inout);      /* one-norm */
    in++;
    inout++;
   }
}
```

Reduction

Example: 1-norm

Step 2: Register the function onenorm with MPI by:

MPI_Op_create((MPI_user_function *)onenorm, commute, &myop);

where

- **onenorm** is the function to be registered.
- **commute** is an input int (scalar) variable which must be set to **1** if the function satisfies the commutative property. Otherwise, it must be set to **0**.
- **myop** is an int (output) to be used in subsequent reduction operation call in place of **onenorm**.

Step 3: Call MPI reduction operation function **MPI_Reduce** using the registered **myop**.

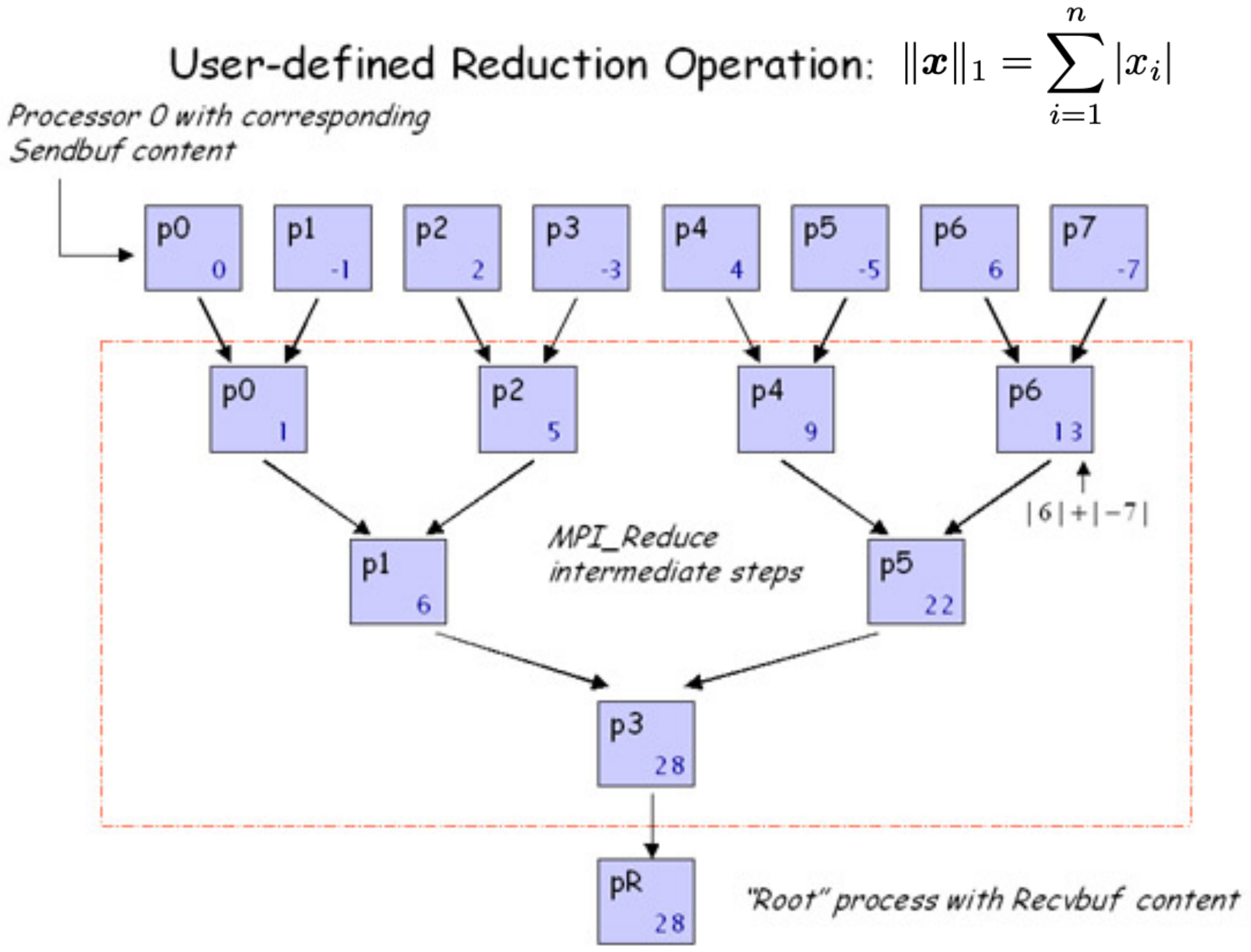
Reduction

Example: 1-norm

```
#include <mpi.h>
#include <stdio>
#include <math.h>

void main(int argc, char* argv[]) {
  int root=0, p, myid;
  float sendbuf, recvbuf;
  MPI_Op myop;
  int commute = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Op_create((MPI_user_function *)onenorm, commute, &myop);
  sendbuf = myid*((int)pow((double)-1,myid));
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_FLOAT, myop,
             root, MPI_COMM_WORLD);
  if (myid == root)
    printf("The operation yields %f\n", recvbuf);
  MPI_Finalize();
}
```

Reduction

Example: 1-norm


User-defined Reduction Operation: $\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|$
Processor 0 with corresponding Sendbuf content
p0 0
p1 -1
p2 2
p3 -3
p4 4
p5 -5
p6 6
p7 -7
p0 1
p2 5
p4 9
p6 13
$|6|+|-7|$
MPI_Reduce intermediate steps
p1 6
p5 22
p3 28
pR 28
"Root" process with Recvbuf content

Nonblocking collective communication

As in the nonblocking point-to-point case, all calls are local and return immediately, irrespective of the status of other processes.

The call initiates the operation, which indicates that the system may start to copy data out of the send buffer and into the receive buffer.

Once initiated, all involved send buffers and buffers associated with input arguments (such as arrays of counts, displacements, or datatypes in the vector versions of the collectives) should not be modified, and all involved receive buffers should not be accessed, until the collective operation completes.

The call returns a **request handle**, which must be passed to a completion call.

Nonblocking collective communication

int MPI_Ibarrier(MPI_Comm comm, MPI_request *request)

int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)

int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)

int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)

References

- http://www.mpi-forum.org

- http://www.open-mpi.org